NCC 2-554
IN-62-CR
289135
99P.

SANTA CLARA UNIVERSITY
Department of Electrical Engineering and Comuter Science

"Performance Analysis of Local Area Networks"

Final Report

Submitted to:

Mr. John Yin, Manager
Communications and Networks Development Branch
NASA-Ames Research Center, MS 233-18
Moffett Field, CA 94035

Prepared by:

Dr. Hasan S. AlKhatib, PI
and
Ms. Mary Grace Hall, RA
Department of EECS
Santa Clara University
Santa Clara, CA 95053

July, 9, 1990

(NASA-CR-186639) PERFORMANCE ANALYSIS OF LOCAL AREA NETWORKS Final Report (Santa Clara Univ.) 99 p CSCL 09B

N90-24767

Unclas
G3/62 0289135

TABLE OF CONTENTS

ABSTRACT

This report describes a simulation of the TCP/IP protocol running on a CSMA/CD data link layer. The simulation was implemented using the simula language, an object oriented discrete event language. It allows the user to set the number of stations at run time, as well as some station parameters. Those parameters are the interrupt time and the dma transfer rate for each station. In addition, the user may configure the network at run time with stations of different characteristics. Two types are available, and the parameters of both types are read from input files at run time. The parameters include the dma transfer rate, interrupt time, data rate , average message size, maximum frame size and the average interarrival time of messages per station.

The information collected for the network is the throughput and the mean delay per packet. For each station , the number of messages attempted as well as the number of messages successfully transmitted is collected in addition to the throughput and mean packet delay per station.

INTRODUCTION

The purpose of this simulation is to allow a user to model a network with specific station parameters to see the characteristics of its behavior. In this way, an existing network may be examined, as well as the effect of any change on the network. For example, to see the effects of adding a station to an existing network without actually doing it, the network could be simulated with the added station using this tool. The simulation could be run with varying characteristics of the added station, and the performance of the network under those conditions could be examined.

To implement this simulation, four basic layers were deliminated. They are the host,tcp,ip,and csmacd. (The host layer basically encompasses user and high level characteristics.) Within each layer, the basic functionality of each was modeled using petri nets. For each layer, a transmitter and a receiver entity were defined. Using the petri nets, the possible states were modeled as well as the events that cause state transitions to occur.

Because simula supports the notion of inheritance, common characteristics were grouped together in progressive levels of definition. For example, each entity requires a buffer to store messages in, whether it is part of the host,tcp,ip,or csmacd layer. Therefore , in implementing the lowest level of this simulation (level_0) the class entity was declared with all the common characteristics defined at that level, including the buffer. Subsequent levels were progressively defined based on preceeding level class declarations until the specific functions for each layer of the network simulated were implemented.

To allow the user to set variables at run time , this simulation is invoked at the topmost level (network level) after prompting the user for the configuration parameters. These parameters are then passed down to the lower levels as needed, or used as global variables when appropriate.

To sum up, it requires eight entities to represent a single station; a receiver and transmitter for each of the host,tcp,ip, and csmacd layers.

DESCRIPTION OF SIMULATION LEVELS

Because simula supports the notion of inheritance,the software was designed in a series of levels, each one defining structures that are common to the next level. This was done to make the software easier to modify and adapt to future changes, for example modeling a different network protocol. It also maintains the commonality of the software, and helps simplify the design.

The levels designed are level_0,level_1,level_2 and the network level. They are described below.

I. LEVEL_0

Level_0 contains the fundamental building blocks of the simulation, the entity and its supporting structures. The process class entity describes the fundamental process structure that all of the station processes require. That is, all the transmitters and receivers of the host,tcp,ip, and csmacd layers.

Each of these processes require a variable state, used to define the current state of the entity (FREE,SENDING,RECEIVING, etc). Upon creation, each entity is set to the FREE state. In addition to the state, each entity also requires a buffer for holding messages or frames and a frame pointer. For ease in programming, a variable return_code is defined to hold the return value of procedure calls.

In addition to the declared variables, the process entity is called with several arguments that define station characteristics. This was done to allow these values to be read from an input file. A title is passed in for use in debugging and future plotting. The id number refers to the id number of the station. Although the id number is common to each of the eight entities that represent a station, it must be passed individually to each entity at the time of creation. It is through the id number that each station knows itself. It is used as an array index. For example, tx_ip(id) would activate tx_csmacd(ip) to pass a frame. The dma_xfer_rate and interrupt_time are also station parameters. These are read from the input file, and passed down to the creation of the entity.

The basic class message is also defined in this layer. It contains all the variables required to collect data about the message for later analysis. The basic definition is that of a link class for manipulation in a queue.

The last significant variable defined in this level is the variable u. This variable is used as a random seed generator.

II. LEVEL_1

Level_1 begins the separation of network layers. In the level, the entity classes host,tcp, ip, and csmacd are defined. All aspects that are common to the transmitter and receiver of each network layer are defined here. For example, both the transmitters and receivers of each layer must receive frames from their adjacent layers, so a procedure receive is defined for each entity class.

In addition to the network layer classes, the class window and tcp_timer are defined for use by the tcp layer. The window controls how many messages may be outstanding at once, and the timer is set whenever a transmission is sent.

III. LEVEL_2

Level_2 defines the structure of the network. It contains includes for all the specific entities that define a station (tx_host,rx_host,etc). These entities are described under IMPLEMENTATION OF NETWORK LAYERS. This layer revolves around the variable num_of_stations, which is passed in as a calling argument. The num_of_stations sets the number of stations for the simulation. The entities that define each station are declared as arrays from one to the num_of_stations. In order to declare a variable size array, the array parameter must be passed in as a calling argument. Besides declaring the arrays of entities that compose the stations, the entities are also instantiated and activated. They are instantiated to allow the creation of their support structures, such as their buffers. The activation allows for the initialization of the entities to their first passivation point in the while true loop, which is where they begin to loop on their subsequent states.

Since the simulation allows for more than one type of station at the user's discretion, stations are created using two for loops. The for loops are controlled by the num_typea_stations and num_of_stations. The first loop from one to num_typea_stations configures stations with the variables dma_xfer_rate and interrupt_time. This represents the creation of type a stations.

The second for loop runs from num_typea_stations + 1 until num_of_stations. This creates stations of type b, using the variables b_dma_xfer_rate and b_interrupt_time. This method results in type a stations having the lowest id numbers. For example, if the user chooses to have twenty stations total and five type a stations, then the stations with id numbers from one to five will be the type a stations and the stations with id numbers from six to twenty will be type b stations.

In addition to the declaration of the entities that comprise the stations, a collection queue is also listed as an include file at this level. The collection queue collects all data used to measure performance.

IV. NETWORK

The network level performs two basic functions: it manages all user interface and calls the simulation. In order to call the simulation, all variables required by the simulation must be set. This is accomplished both by absorbing the user inputs and calculating certain key variables. The network level also controls the length of the simulation, based on the number of messages successfully transmitted per station.

The user interface is composed of two parts, prompting the user and writing data collected to an output file. The user is prompted for the number of stations and their type. To choose the type of station, three options are offered: type_a stations, type_b stations or a mix of the two. If a mix of station types is chosen the user is again prompted for the number of type_a stations. If the user chooses to have more type_a stations than the number of stations, the value is automatically reset to the number of stations. No other error checking is performed on the user input.

After the number and type of stations have been defined, the input files are read to obtain the values of station characteristics. There are two station types currently implemented, and the corresponding values of these types are contained in the file " station_typea" and "station_typeb". The names of the input files are directly referenced in the network layer. To add another input file containing station characteristics, the network layer would require modification. The modifications required would be the addition of the new input file, and the definition of new variables for the dma_xfer_rate and interrupt_time, which are currently the only difference between the station types. The new variables would be passed as calling arguments to the simulation.

The format of the input files is identical, regardless of the station type they define.The input files are read using the read_input routine, which assigns the values in the file to the corresponding variables. For this reason, the order of the values in the file is crucial for correct assignment. The read_input routine reads a single value from each line of the input file. The variables read in their assigned order are data_rate, interrupt_time, dma_xfer_rate, aver_arrival_time, prop_delay, max_frame_size, and aver_msg_size. All the values are real except the last two, which are integer.

The output file is written at the end of the simulation. It contains information on the throughput and average delay per frame for the individual stations and the whole network. Also included is the number of successful and rejected messages per station as well as the simulation time required. All values are calculated in the collection_queue routines. The output is written to a file "output" using the print_data routine. The file name is directly referenced in that routine. A modification to the output file would require changes to be made to the print_data routine, and the collection_queue routines that calculate the data.

After the input files have been read,the value of tau is calculated. The simulation is called with the value of tau and all variables read from the input files as calling arguments. The simulation then loops until each station achieves a minimum number of successful transmissions. The data is then collected, first on a per station basis and then for the network as a whole. It is then printed, and the program terminates.

IMPLEMENTATION OF NETWORK LAYERS

Each of the layers described below represents a network level protocol. They are represented by both a transmitter and receiver entity concerned with the passage of messages. The various states of the entities are listed,along with events that cause a state transition.

I. IMPLEMENTATION OF HOST LAYER

The host layer of this simulation represents all upper layers of network protocol above the tcp layer. In this layer the arrival of new messages is generated and their final reception occurs. This is accomplished by use of message queues, one for the transmitter and one for the receiver.

HOST TRANSMITTER

The host transmitter has no defined states. It loops on generating messages. After a message is generated, the host transmitter checks to see if the tcp transmitter is idle. If it is, it activates it, holds for an interrupt time, and then loops back to generate the next message.

The generate_messages procedure of the host transmitter is responsible for setting message parameters and their arrival rate. It begins by using the global variables aver_arrival_time and u to get a random arrival time. It holds for this amount of time before proceeding to simulate a poisson arrival rate.

The queue length is checked next to see if it is less than MAX_HOST_MESSAGE_COUNT. This is done because of the memory constraints imposed running the simulation. It is desirable to have no restrictions on the number of messages the host transmitter may generate,but since the arrivals of the messages is a poisson process, it has no memory of past arrivals. So the modeling of the arrivals remains valid.

If there is room in the message queue, a new message is created. The time of creation is noted for bookkeeping purposes. A random destination other than its own address is generated. The size of the message is assigned using the global variable aver_msg_size. The message is placed in the queue so that the tcp transmitter may process it. The host transmitter then loops back around to generate the next message.

HOST RECEIVER

The host receiver has two states: FREE and RECEIVING. The FREE state is the idle state, when the host receiver is ready to process the next message. The receiver is capable of receiving more than one message in its queue at once, as compared to the lower layers that process one frame at a time.

When a message arrives, a state transition to RECEIVING occurs. The message is taken out of the queue and discarded, since all bookkeeping regarding the message is done by the tcp receiver. The state then transitions back to free after holding for an interrupt time.

II. IMPLEMENTATION OF TCP LAYER

The tcp layer is composed of two entities, the transmitter and the receiver as well as a number of supporting structures for the processing of messages. On the transmitter side this layer receives messages from the host, and breaks them into frames for transmission. On the receiving side, it re-assembles the frames back into messages and passes them to the host. The tcp layer is responsible for determining if frames sent have been received. Positive acknowledgements are used to accomplish this. A buffer and a window are used to process messages. Although each of these has a maximum size, if there is not enough room to process the messages, this upper limit is simply increased to accomodate them. To decrease the number of transmissions across the network, acknowledgements are piggybacked onto data packets whenever possible.

TCP TRANSMITTER

The tcp transmitter has three states: FREE, RECEIVING, and SENDING. The FREE state is the idle state, with no messages to process or packets to send. If there are any messages from the host layer of that station to transmit, a state transition to the RECEIVING state occurs, provided that the buffer of the tcp transmitter is not full.

In this state, messages are removed from the host message queue and packetized for transmission. They are also added to a collection queue for easy data retrieval when the transmission is completed.

The tcp transmitter removes messages directly from the host message queue of that station. In addition to transmitting messages, it also transmits acknowledgements and re-transmits packets that have timed out. If there are any acknowledgements, time-outs, or data packets to send a state transition to the SENDING state occurs. In this state, whenever a transmission is sent a timer is set. Should the timer expire before an acknowledgement for the transmission is received, the transmission is resent and the timer reset.

Transmissions are sent on a priority basis.The message with the highest priority is an acknowledgement, followed by a re-transmission of a message that has timed out and lastly a data packet.

TCP RECEIVER

The tcp receiver has two states: FREE and RECEIVING. The FREE state is the idle state, when the tcp receiver is available to receive and has no frame to process. In this state, it activates the ip receiver in case it has any frames to send and then passivates.

When a frame arrives, a state transition to RECEIVING occurs. The type of the frame is checked to see if it is a data frame and/or an acknowledgement frame. If it is an acknowledgement frame, the procedure ack_message of the transmitter is called to process it. A data frame is processed by the receiver.

Next the frame is removed from the window and the buffer, and the sizes are adjusted accordingly. If the frame is the last data frame of a message, the message is re-assembled and forwarded to the host receiver. The host receiver is then activated to process the message.

III. IMPLEMENTATION OF IP LAYER

The ip layer implementation is composed of two entities, the receiver and transmitter. It is a very simple layer that receives a pointer to the buffer contents and passes it to the next appropriate layer. Both entities receive packets through their receive routines. In this layer, the frame pointer is checked directly rather than the state of the buffer. Only one packet is required for the entity to activate itself.

IP TRANSMITTER

The ip transmitter has three possible states: FREE, READY, and SENDING. FREE represents the idle state, with no frame in the buffer. When a frame is received, a state transition to the READY state occurs. At this stage, the ip transmitter is ready to transmit a frame to the csmacd layer. It calls the receive routine of the csmacd transmitter for that station, and attempts to pass the frame. It loops in that state until the frame is successfully passed to the csmacd transmitter.

When the frame is successfully passed, the state transitions to SENDING. At this point the ip transmitter activates the transmitter entities of the tcp and csmacd layers for that station, and resets itself.

The ip transmitter also allows for a failed transmission. If the csmacd layer of that station is unable to transmit the frame, it calls the routine transmit_failed. This routine in turn calls the transmit_failed routine of the tcp transmitter of that station.

IP RECEIVER

The ip receiver has two states: FREE and BUSY. The FREE state corresponds to the idle state, when the receiver is available to receive and has no current frame. In this state, the receiver of the csmacd layer for that station is activated in case it has a frame to send to the ip receiver. If the ip receiver receives a frame, it transitions to the BUSY state. It loops in that state until it successfully transmits the frame to the tcp receiver. Whether the transmission is successful or not, it activates the receiver of the tcp layer to allow any possible state transitions of that layer to occur.

After a successful transmission to the tcp receiver, the frame pointer is reset to none and the state is set to FREE.

IV. IMPLEMENTATION OF CSMA/CD LAYER

The csmacd layer implementation is based on the petri net model of the csmacd. It is composed of two entities, the receiver and transmitter as well as the definition of a class channel_csmacd. The entities are processes that have the capability of activating and passivating themselves. Class channel_csmacd, on the other hand, is a simple class definition that basically consists of a queue. The buffer for the csmacd layer is actually implemented as a characteristic of the csmacd receiver and transmitter. That is to say, both the transmitter and the receiver each have their own buffer.

Both entities receive packets through their receive routines, which check to see if the buffer is available. If it is,it accepts the incoming packet and sets the buffer state to FULL. Only a single packet is required to set the buffer state to FULL. If the buffer is FULL when an incoming packet arrives, it is ignored.

Each csmacd entity is designed as a forever true loop. If there is an incoming message in the buffer or a state transition has occurred, it loops determining through the next state procedure if another transition can occur. When all possible state transitions have been exhausted, it passivates. To be awakened, the csmacd transmitter must be called by the ip transmitter. The csmacd receiver, on the other hand, may be awakened either by a sending csmacd transmitter or by its ip receiver layer.

For each possible state shown on the petri net, there is a subroutine. The state subroutines are contained in csmacd_txs for the transmitter, and csmacd_rxs for the receiver. These subroutines check for potential transitions, and return a boolean value of true if a transition of state occurs.

CSMA/CD TRANSMITTER

The csmacd transmitter has five possible states: DISABLED, FREE, WAIT_FOR_RE_TX, ATTEMPT_TX,and ACQUIRE_CHANNEL. Each of these states will be described below.

The FREE state represents the idle state, when the transmitter is not engaged. In this state, for a transition to occur, the buffer must be full and the channel must be free. If this is the case, the state transitions to ATTEMPT_TX. If the buffer is full, but the channel is not free the transmitter waits in the channel queue until the channel becomes free.

The ATTEMPT_TX state represents the transmitter attempting to transmit a packet. In this state, the transmitter holds for a peroid of tau, and then checks the channel_state variable to see if it is zero. If it is, it increments the channel_state variable and holds for another tau . It then checks to see if the channel_state variable is greater than one. If it is, a collision has occurred. If not, it ends the collision window and transitions the state to ACQUIRE_CHANNEL. If a collision occurs, the state is set to DISABLED. So, from the ATTEMPT_TX state there is always a state transition to either ACQUIRE_CHANNEL or DISABLED.

The DISABLED state represents the period when a station realizes it has has collided with one or more stations and must jam transmissions. Each station involved in a collision will independently realize the event by checking the channel_state variable. In this way, the channel remains a passive element and the individual stations are the active determinants of their state.

When a station realizes it is disabled, it holds for a jam time of two tau and then changes state to WAIT_FOR_RE_TX. It decrements the channel_state variable and checks to see if the variable has returned to zero. This indicates that the jam is over, and all stations involved are ready to wait for re-transmission. If the jam is over, the station will check the channel queue to see if there were any stations that were waiting for the channel. If so, it wakes them up. In this way, preference for new transmissions over transmissions involved in a collision is preserved.

In the WAIT_FOR_RE_TX state, the station checks to see if it has attempted to send the transmission more than max_tries (sixteen attempts). If it has, it retrieves the frame from the buffer and calls the transmit_failed routine of the station's ip layer. The csmacd transmitter is reset for the next transmission and the ip transmitter is activated.

If the csmacd transmitter has not exceeded max_tries, it increments a binary backoff variable n, and generates a random number in the window of 2**n - 1. It holds for that time times tau, and then transitions back to the FREE state to attempt to transmit again.

The ACQUIRE_CHANNEL state represents the station transmitting. In this state, the collision window has ended and the station definitely has the channel. The transmission time is calculated based on the bits per packet divided by the data rate. The transmitter simply sends the message by calling the receive routine of the destination csmacd receiver, and then activating it. No attempt is made to determine if the transmission is successfully received. The csmacd receiver is then reset (the buffer is empty and the state is FREE) and the ip transmitter is activated.

CSMA/CD RECEIVER

The csmacd receiver has three possible states: DISABLED, FREE, and RECEIVING. Because it is impossible for a receiver to receive an unsuccessful transmission in this simulation, only the FREE and RECEIVING states are modeled.These states are described below.

The FREE state represents the idle state of the csmacd receiver. In this state, the receiver checks to see if the buffer is full. If it is, it resets the state to receiving. If not, it does nothing.

The RECEIVE state attempts to pass the buffer contents to its ip receiver.It calls the ip receive routine to pass the buffer contents, and holds for rx_delay. The ip receiver is activated, and the csmacd receiver is reset (the buffer is empty and the state is FREE) . If the csmacd receiver is unsuccessful in attempting to pass its buffer contents to the ip receiver, it passivates until re-awakened by the ip receiver or a csmacd receiver.

CSMA/CD CHANNEL

The csmacd channel is not a process class, but a simple class definition. It is composed of a queue and a class state variable. The queue is used for stations wishing to transmit when the channel is already in use. When the channel becomes free (after a successful transmission or at the end of a jam) the stations in the queue are all activated at once and allowed to contend for channel acquisition. If more than one station is in the queue, this generally results in a collision with normal contention resolution. In this way, the behavior of the csmacd protocol is preserved.

Access to the channel is controlled by the channel state variable. If the channel state variable is zero, the channel is free and a station may transition state to ATTEMPT_TX. If it is not zero, the station enters the channel queue.

To simulate the collision window, two possible cases are considered. After a station has transitioned states from FREE to ATTEMPT_TX, it holds for a period of tau. It next checks the channel state variable to see if it is still zero. If it is, no collision has occurred yet and it continues to attempt to gain the channel. If it is not, the station has collided with another station and it holds for another tau and follows the collision protocol.

If no collision occurs within the first tau period, the station attempting to transmit increments the channel state variable and holds for another tau. It then rechecks the channel state variable to see if it is greater than one. If it is, another station is also attempting to transmit and a collision has occurred. If not, it has successfully gained the channel.

At the end of a successful transmission, the station decrements the channel state variable and it returns to zero, thus leaving the channel in a free state. In the case of a collision, as each station completes the hold for the jam time, it decrements the channel state variable and checks to see if it has returned to zero. If it has, all stations involved in the collision have now completed their jam and the channel is free. Any stations in the channel queue are now awakened to compete for the channel. If the channel state variable is not zero, all station involved in the collision have not completed their jam and the channel is not free.

MAKEFILE DESCRIPTION

A makefile is available to compile all code required by the simulation. By typing make, an executable called network will be built. Since the system file that contains the grammar rules used by make does not include the simula language, all dependencies and rules for construction must be explicitly stated. This is difficult to implement in simula, since compiling a ".sim" file results in the creation of a ".o" file and a ".s" file. The ".o" file contains the object code, and the ".s" file is used by the assembler phase of compilation. To make use of the make utility, it was decided to state dependencies based on the ".o" files alone, since the ".s" files are automatically generated with the compilation anyway.

To add a new simula file to the list of files to be compiled, the following format should be used:

```
new.o:     new.sim
           $(SIM_COMP) $*
           $(SIM_ASM) $*.o  $*.s
```

To add a simula file that will be used as an include file in another simula file, the dependency is listed directly:

```
old.o:     new.sim  \
           other_include.sim
```

The above steps will insure that the dependencies are always correct, so that recompilation of the appropriate files is automatic after any modifications.

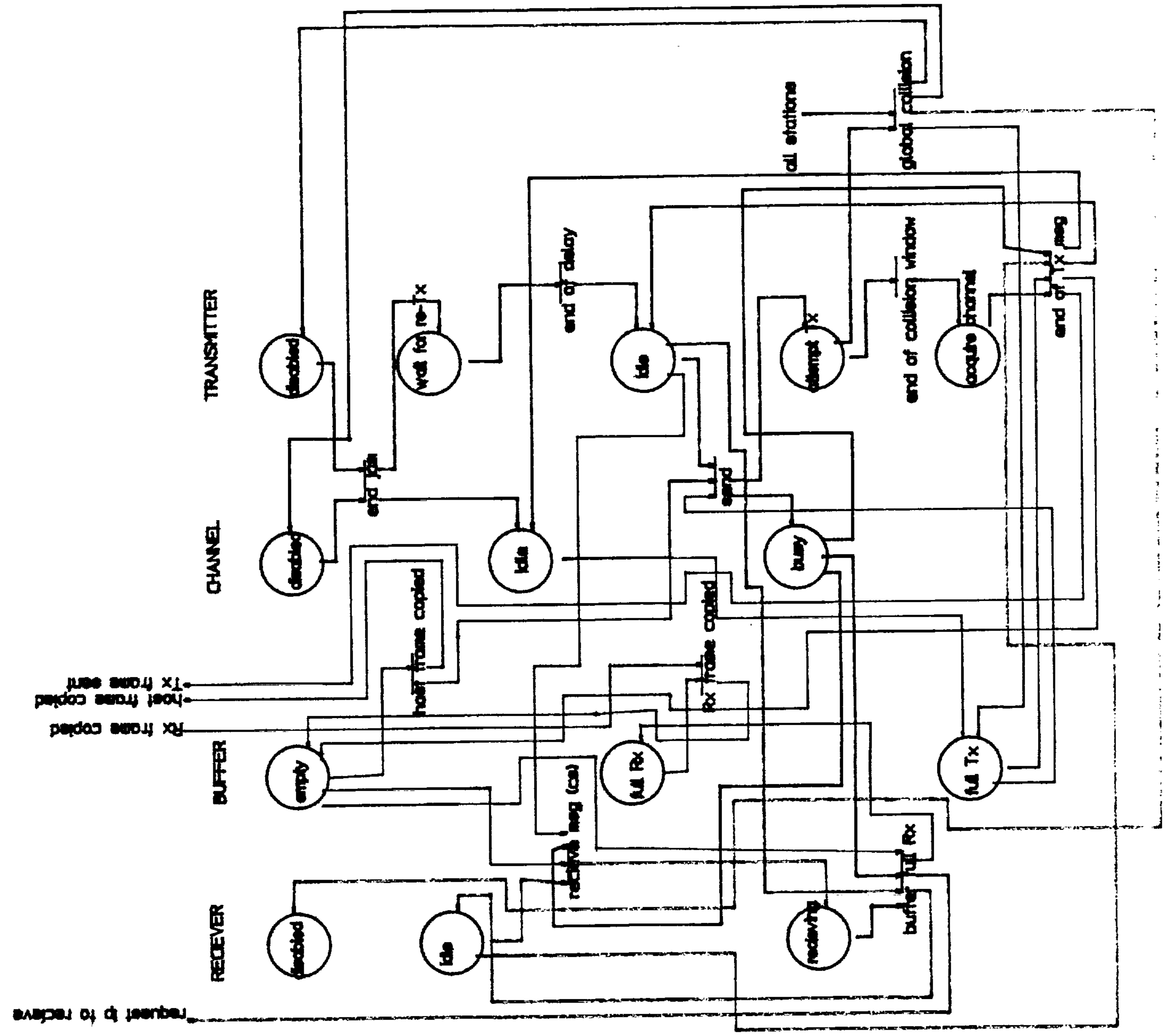
The makefile contains two targets, the network executable and a debug executable. To use the debugger, code must be compiled with the debug option and thus it is included. To make a debug executable, type " make dnetwork". This will compile the code with a debug option.

When one simula file is used as an include file in another simula file, no separate ".o" file is created for the included ".sim" file. Instead, the simula code in the include file is incorporated in the ".o" file generated for the master file. That is why the included ".sim" file is listed as a dependency for the master ".sim" file rather than its ".o" file that one might think should be created.

CONCLUSION

In its current state of implementation, this simulation models only LAN composed of stations. A logical progression would be to implement bridges to allow for a more complex LAN. This could be accomplished by creating another layer above the network layer to call the simulation. Since the network layer as currently implemented is a simulation call, it would have to be modified so that there is a single simulation running rather than a series of network simulations.

Another area of future development is the user interface. Currently, a simple menu is presented to the user and they type in the letter corresponding to their choice of station type. As the user definable parameters increase, a graphical menu interface may be more appropriate.



ORIGINAL PAGE IS
OF POOR QUALITY

```
!                   SANTA CLARA UNIVERSITY

                    DEVELOPED FOR NASA/AMES

                    NCC2-554

                    PERFORMANCE ANALYSIS OF LAN      ;
!*************************************************;

!
         this file contains the lowest level declarations
required in this simulation. Most of the following classes
are either inherited or referenced by succeeding classes.;

!*************************************************;
SIMULATION CLASS level_0;
BEGIN

  INTEGER  u;                                        ! seed number for random
                                                             number generator;

  HEAD CLASS message_queue(title);
  VALUE title; TEXT title;
  BEGIN
  END;


  LINK CLASS dataunit;
  BEGIN
  INTEGER bytes, dest_addr, source_addr;
  END;

  dataunit CLASS message;
  BEGIN
  INTEGER type, id;
  long real createtime, donetime;
  long real time_per_message;
  BOOLEAN rejected;

        createtime := 0.0;
        donetime   := 0.0;
        time_per_message  := 0.0;
        rejected   := FALSE;
  END;

  message CLASS frameunit;
  BEGIN
  REF(message) msgptr;
  INTEGER seqnum, acknum, setwindow;
  BOOLEAN  ack,  fin;

          ack := FALSE;
          fin := FALSE;
  END;

 HEAD CLASS buffer;
 BEGIN
 REF(frameunit) front, tail, current, msgtail;   !buffer ref. frame pointers;
   INTEGER maxsize, cursize;
   INTEGER state;
```

```
        front :- NONE;
        tail :- NONE;
        current :- NONE;
        msgtail :- NONE;
END;

PROCESS CLASS timer;
BEGIN
INTEGER status;

END--of--timer;

PROCESS CLASS entity(title,id,dma_xfer_rate,interrupt_time);
VALUE title; TEXT title;
INTEGER id;
long real dma_xfer_rate;
long real interrupt_time;
BEGIN
  REF(buffer) buf;
  REF(frameunit) frame;
  INTEGER state;
  INTEGER return_code;


 !**********************************************************;
 !;
 !;
 !               procedure reset;
 !
 !        called by: specific process reset routines;
 !;
 !        calls: none;
 !;
 !        returns: none;
 !;
 !        globals used: ;
 !                state := FREE( set to 0 );
 !                frame := none;
 !;
 !        actions: ;
 !                resets the entity variables;
 !;
 !**********************************************************;
 procedure reset;
 begin
        ! reset the variables;
        state := 0;             ! this sets it to free;
        frame :- none;

 end;


    ! main body, set variables;
    begin
        ! get buffer,set buffer state to EMPTY;
        buf :- new buffer;
        buf.state := 0;
        ! set state of entity to FREE
        state := 0;
    end;

 END***entity***;
```

```
  ! main program of simulation;
  BEGIN
        u := 1000003;                 ! initialize seed;

  END***MAINPROGRAM***;

END***SIMULATION***;
```

```
!                    SANTA CLARA UNIVERSITY

                     DEVELOPED FOR NASA/AMES

                     NCC2-554

                     PERFORMANCE ANALYSIS OF LAN      ;
!*****************************************************;

!level_1.sim - subclass of level_0                    ;
!
                     this file inherits allknowledge of
         level_0 simulation and declares classes that
         define common characteristics of the layers
         modeled in this simulaton.;

!*****************************************************;
external CLASS level_0;

level_0 CLASS level_1;
BEGIN
%INCLUDE define_def.sim

entity CLASS host;
BEGIN
ref(message_queue) msg_queue;
i ~f(message) host_msg;


END++OF++HOST;


entity CLASS tcp;
BEGIN
REF(window) win;
REF(frameunit) lastsent;
REF(timeout_queue) timeoutq;
BOOLEAN timed_out, next_msg;
INTEGER timeout_count;
INTEGER buf_space, last_byte_sent;

integer procedure receive(recvframe);
REF(frameunit) recvframe;
!-------------------------------------------------------------;
! procedure receive                                           ;
!                                                             ;
! This procedure handles the addition of frames to the tcp    ;
! buffer.  If there is room in the buffer the frame is        ;
! added and the buffer parameters and pointers are updated,   ;
! otherwise the receive fails.                                ;
!                                                             ;
! Returns: OK, FAILED                                         ;
                                                              ;
! Globals: buf (current buffer)                               ;
!-------------------------------------------------------------;
begin
integer rc;
    if buf.state = EMPTY then
    begin             ! buffer is empty, receive the frame & update pointers ;
        recvframe.INTO(buf);
        buf.front :- recvframe;
        buf.current :- recvframe;
        buf.tail :- recvframe;
        buf.state := NENF;  ! update the state of the buffer                    ;
        next_msg := TRUE;   ! set flag to indicate message ready to send        ;
        rc := OK;           ! set the return code to indicate success           ;
        go to Return;       ! processing complete, leave this procedure         ;
    end
    else if buf.state = NENF then
    begin              ! buffer is not empty, not full, check size of frame     ;
        if (buf.maxsize >=  buf.cursize + recvframe.bytes) then
        begin       ! okay to receive frame, receive it & perform updates       ;
            recvframe.INTO(buf);
            buf.tail :- recvframe;
            if (buf.current == NONE) then
            begin
                buf.current :- recvframe;
            end;
            rc := OK;
            go to Return;
        end else begin  ! no room for frame, set buffer to FULL & fail receive;
            buf.state := FULL;
            rc := FAILED;
            go to Return;
        end;
    end
    else if buf.state = FULL then
    begin              ! buffer FULL , receive fails                             ;
        rc := FAILED;
        go to Return;
    end;
Return:
begin
    receive := rc;
    if (rc = OK) then
    begin              ! if status okay, update the buffer's current size       ;
        buf.cursize := buf.cursize + recvframe.bytes;
    end;
end;
end--of--receive;

procedure reset_buffer;
!-----------------------------------------------------------------------;
! procedure reset_buffer                                                ;
!                                                                       ;
! This procedure reset all the buffer parameters to their               ;
! initial (empty) state.                                                ;
!                                                                       ;
! Globals: buf (current buffer)                                         ;
!-----------------------------------------------------------------------;
begin
        buf.front :- NONE;
        buf.current :- NONE;
        buf.tail :- NONE;
        buf.state := EMPTY;
        buf.cursize := 0;
end;

integer procedure outofbuf(msg_id);
INTEGER msg_id;
!-----------------------------------------------------------------------;
! procedure outofbuf
!                                                                    ;
! This procedure removes all the frames of a given message          ;
! from the tcp buffer and performs the required updates             ;
  to keep the buffer current.                                        ;
!                                                                    ;
! Returns: OK                                                        ;
!                                                                    ;
! Globals: buf_space, buf, next_msg, return_code                     ;
!--------------------------------------------------------------------;
begin
    REF(frameunit) outofptr, currentptr;
    integer count;
    count := 0;
    currentptr :- buf.FIRST;    ! start at the beginning of the buffer(buf)   ;
    return_code := FAILED;

    while currentptr=/=NONE do  ! search until end of buf reached             ;
    begin
        count := count +1;      ! trace through the loop for debug            ;
        if currentptr.id=msg_id then
        begin                   ! take all matching id #'s out of buf         ;
            outofptr :- currentptr;
            currentptr :- (currentptr).SUC;
            outofptr.OUT;
            return_code := OK;
            buf_space := buf_space + outofptr.bytes;
        end else
        begin                   ! no match, update current pointer to next    ;
            currentptr :- (currentptr).SUC;
        end;
    end;
    if (return_code = OK) then
    begin
        buf.front :- buf.FIRST; ! update pointers if frames removed from buf ;
        if (buf.front =/= NONE) then
        begin
            next_msg := TRUE;   ! if another message in buf, set flag to send;
        end else
        begin
            reset_buffer;       ! no more frames to send, reset the buffer    ;
        end;
    end;
    outofbuf := return_code;
end--of--outofbuf;

boolean procedure reserve_buffer_space(addbytes);
!--------------------------------------------------------------------;
! procedure reserve_buffer_space                                     ;
!                                                                    ;
! This procedure checks the space in the buffer to see if           ;
! the entire message will fit.  If so, the space is reserved;
! and TRUE is returned, else FALSE is returned.  This               ;
! prevents partial messages in the buffer.                           ;
!                                                                    ;
  Returns: TRUE - success, FALSE - failure                           ;
                                                                     ;
! Globals: buf (current buffer), buf_space                           ;
!--------------------------------------------------------------------;
integer addbytes;
begin
    integer bytes_left;
    bytes_left := buf_space - addbytes;
    if (bytes_left >= 0) then
    begin
        buf_space := bytes_left;
        reserve_buffer_space := TRUE;
    end else begin
        buf.state := FULL;
        reserve_buffer_space := FALSE;
    end;
end;

integer procedure set_donetime(msgptr);
!-------------------------------------------------------------;
! procedure set_donetime                                      ;
!                                                             ;
! This procedure sets the donetime parameter of the message  ;
! to indicate that the message has been sent and              ;
! acknowledged.  This is done for data collection statistics;
! purposes.                                                   ;
!                                                             ;
! Returns: OK, FAILED                                         ;
!                                                             ;
! Globals: buf (current buffer)                               ;
!-------------------------------------------------------------;
REF(message) msgptr;
begin
    if msgptr=/= NONE then  ! check for valid message pointer                ;
    begin
        msgptr.donetime := TIME;  ! set the current time to message donetime ;
        set_donetime := OK;       ! status of procedure is okay              ;

    end else begin
        set_donetime := FAILED;
    end;
end;

begin           !main;
                ! set up parameters and instantiate window;
    win :- new window;
    win.state := EMPTY;
    win.maxsize := 10240;
    win.cursize := 0;
    timeoutq :- new timeout_queue;
    timeout_count := 0;
    timed_out := FALSE;
    next_msg := FALSE;
end--of--main;

END++OF++TCP;


entity CLASS ip;
BEGIN
BOOLEAN csmacd_rc;

[  lean procedure receive(recvframe);
REF(frameunit) recvframe;
begin
    if (state = FREE) and (frame == NONE) then
    begin
        state := READY;
        frame :- recvframe;
    end else begin
        receive := FALSE;
    end;
  );

END;


entity CLASS csmacd;
BEGIN
    boolean change_state;          ! flag for change of state;
    boolean buffer_interrupt;      ! flag for frames in the buffer;

!***********************************************************;
!;
!;
!    ref( frameunit ) procedure get_frame_from_buffer;
!;
!    called by: csmacd_txs_acquire_channel,csmacd_rxs_idle ;
!;
!    calls: none;
!;
!    returns: ;
!        a pointer to the frame from the buffer;
!        if the buffer is empty,pointer to none;
!;
!    globals used: ;
!        buf - the current buffer;
!;
!    actions: ;
!        takes a frame out of the buffer;
!        should set the buffer state to empty;
!        does not do so because of integrity of;
!        buffer during transmission,i.e., buffer;
!        should not be released until frame is ;
!        transmitted;
!;
!***********************************************************;
ref(frameunit) procedure get_frame_from_buffer;
begin
ref(frameunit) out_frame;

     ! initialize the variables;
     out_frame :- none;

     ! if the buffer isn't empty,get the frame out;
     if ( buf =/= none) then
     begin
     if ( not ( buf.empty )) then
     begin
         ! get the frame from the buffer ;
         out_frame:- buf.first;
         out_frame.out;

         ! set the buffer state;
         !buf.state := EMPTY;

     end;
     end;
! set the return variable;
get_frame_from_buffer :- out_frame;
```

```
end;

!*****************************************************;

!;
!       procedure clear_buffer;
!;
!   called by: reset ;
!;
!   calls: none;
!;
!   returns: none;
!;
!   globals used: ;
!       see below;
!;
!   actions: ;
!       resets the csmacd buffer variables;
!;
!*****************************************************;
procedure clear_buffer;
begin
    ! set variables back to zero;
    buf.state := FREE;

    ! reset buffer interrupt;
    buffer_interrupt := false;

    ! reset the frame pointers;
    buf.clear ;
end;


    ! main body for csmacd;
    begin
        change_state := false;
        buffer_interrupt := false;
    end;
END;

  CLASS window;
  BEGIN
  REF (dataunit) current, front, back;
  INTEGER maxsize, cursize;
  INTEGER state;
  INTEGER rc;       !rc - return code;

  integer procedure rxtcp_outof(frameptr);
  REF(frameunit) frameptr;
  begin
    cursize := cursize - frameptr.bytes;
    if (cursize > 0 ) then
    begin
        state := NENF;
        front :- front.SUC;
    end else begin
        reset_window;
    end;
    rxtcp_outof := OK;
  end--of--rxtcp_outof;
```

```
[illegible]

REF(frameunit) frameptr;
begin
end---of---txtcp_outof;

procedure reset_window;
begin
  state := EMPTY;
  cursize := 0;
  front :- NONE;
  current :- NONE;
  back :- NONE;
end;

boolean procedure reserve_space(bytes);
integer bytes;
begin
  if maxsize > (cursize + bytes) then
  begin
      cursize := cursize + bytes;
      reserve_space := TRUE;
  end else begin
      reserve_space := FALSE;
  end;
end;

procedure cancel_reserve_space(bytes);
integer bytes;
begin
  cursize := cursize - bytes;
end;

boolean procedure addto(frameptr);
REF(frameunit) frameptr;
begin
  if state = FULL then
  begin
      addto := FALSE;
      go to Return;
  end;
  if state = NENF then
  begin
      if maxsize >= cursize  then
      begin
           state := NENF;
           current :- frameptr;
           back :- frameptr;
           addto := TRUE;
           go to Return;
      end;
      if maxsize < cursize then
      begin
           state := FULL;
           addto := FALSE;
           go to Return;
      end;
  end;
  if state = EMPTY then
  begin
      state := NENF;
      front :- frameptr;
      current :- frameptr;
      back :- frameptr;
```

```
          addto := TRUE;
          go to Return;
     end;
?   :urn:
end;

begin                !*** window - main program ****!;
  state := 0;
end;
END++OF++WINDOW;

HEAD CLASS timeout_queue;
begin
end;

LINK CLASS timeout_unit;
begin
     REF(tcp) layer;
     REF(tcp_timer) timerptr;
     REF(frameunit) frameptr;
     long real time_up;
     integer event;
     integer status;

     status := NOT_SET;
end;

timer CLASS tcp_timer;
BEGIN
   (timeout_unit) tcp_event;
long real currtime;

     procedure setup(timeout_event);
     REF(timeout_unit) timeout_event;
     begin
          tcp_event :- timeout_event;
          tcp_event.status := SET;
     end;

     procedure start;
     begin
          hold(tcp_event.time_up);
          timeout_occurred;
     end;

     procedure timeout_occurred;
     begin
          (tcp_event.layer).timeout_count := (tcp_event.layer).timeout_count + 1;
          tcp_event.status := FRAME_TIMED_OUT;
          (tcp_event.layer).timed_out := TRUE;
          reactivate(tcp_event.layer);
     end;

 begin
      start;
 end;
 END;


 ! main body;
 %INCLUDE define_assign.sim
 END.
```

```
!                SANTA CLARA UNIVERSITY

                 DEVELOPED FOR NASA/AMES

                 NCC2-554

                 PERFORMANCE ANALYSIS OF LAN     ;
!**************************************************;

!       this level inherits the characteristics of
        both level_0 and level_1 . It specifies the
        exact nature of all entities modeled.
;
!**************************************************;
external CLASS level_1;
level_1 class level_2(num_of_stations,tau,prop_delay,interrupt_time,
                        data_rate,dma_xfer_rate,aver_arrival_time,
                        max_frame_size,aver_msg_size,b_interrupt_time,
                        b_dma_xfer_rate,num_typea_stations);
integer num_of_stations;
integer tau;
long real prop_delay;
long real interrupt_time;
long real data_rate;
long real dma_xfer_rate;
long real aver_arrival_time;
integer max_frame_size;
integer aver_msg_size;
long real b_dma_xfer_rate;
long real b_interrupt_time;
integer num_typea_stations;

BEGIN
integer msg_count;                ! unique message id number;


%INCLUDE collection_queue.sim
%INCLUDE rx_host.sim
%INCLUDE tx_host.sim
%INCLUDE tcp.sim
%INCLUDE ip.sim
%INCLUDE tx_csmacd.sim
%INCLUDE rx_csmacd.sim
%INCLUDE channel_csmacd.sim


ref(rx_csmacd) array srx_csmacd(1:num_of_stations);
ref(tx_csmacd) array stx_csmacd(1:num_of_stations);
ref(channel_csmacd) schannel_csmacd;

REF (tx_tcp)    array stx_tcp(1:num_of_stations);
REF (tx_ip)     array stx_ip(1:num_of_stations);
REF (rx_host)   array srx_host(1:num_of_stations);
REF (rx_tcp)    array srx_tcp(1:num_of_stations);
REF (rx_ip)     array srx_ip(1:num_of_stations);
REF (tx_host)   array stx_host(1:num_of_stations);
REF (collection_queue)    collectionq;  ! collection queue for network;

 INTEGER i;
```

```
BEGIN
        ! initialize variables;


        for i := 1 step 1 until num_typea_stations do
        begin
                ! create entities;
                srx_host(i) :- new rx_host("rx_host",i,
                                         dma_xfer_rate,interrupt_time);
                stx_tcp(i) :- new tx_tcp("tx_tcp",i,
                                         dma_xfer_rate,interrupt_time);
                srx_tcp(i) :- new rx_tcp("rx_tcp",i,
                                         dma_xfer_rate,interrupt_time);
                srx_ip(i) :- new rx_ip("rx_ip",i,
                                         dma_xfer_rate,interrupt_time);
                stx_ip(i) :- new tx_ip("tx_ip",i,
                                         dma_xfer_rate,interrupt_time);
                srx_csmacd(i) :- new rx_csmacd("rx_csmacd",i,
                                         dma_xfer_rate,interrupt_time);
                stx_csmacd(i) :- new tx_csmacd("tx_csmacd",i,
                                         dma_xfer_rate,interrupt_time);
                stx_host(i) :- new tx_host("tx_host",i,
                                         dma_xfer_rate,interrupt_time);
        end;

        ! configure type b stations;
        for i := (num_typea_stations + 1) step 1 until num_of_stations do
        begin
                ! create entities;
                srx_host(i) :- new rx_host("rx_host",i,
                                         b_dma_xfer_rate,b_interrupt_time);
                stx_tcp(i) :- new tx_tcp("tx_tcp",i,
                                         b_dma_xfer_rate,b_interrupt_time);
                srx_tcp(i) :- new rx_tcp("rx_tcp",i,
                                         b_dma_xfer_rate,b_interrupt_time);
                srx_ip(i) :- new rx_ip("rx_ip",i,
                                         b_dma_xfer_rate,b_interrupt_time);
                stx_ip(i) :- new tx_ip("tx_ip",i,
                                         b_dma_xfer_rate,b_interrupt_time);
                srx_csmacd(i) :- new rx_csmacd("rx_csmacd",i,
                                         b_dma_xfer_rate,b_interrupt_time);
                stx_csmacd(i) :- new tx_csmacd("tx_csmacd",i,
                                         b_dma_xfer_rate,b_interrupt_time);
                stx_host(i) :- new tx_host("tx_host",i,
                                         b_dma_xfer_rate,b_interrupt_time);
        end;

        for i := 1 step 1 until num_of_stations do
        begin

                ! activate the entities;
                activate srx_csmacd(i) ;
                activate stx_csmacd(i) ;
                activate srx_ip(i) ;
                activate stx_ip(i) ;
                activate srx_tcp(i) ;
                activate stx_tcp(i) ;
                activate srx_host(i) ;
                activate stx_host(i) ;
        end;
        ! declare a channel;
        schannel_csmacd :- new channel_csmacd;
        collectionq :- new collection_queue;
        msg_count := 1;
END;
END;
```

```
!                SANTA CLARA UNIVERSITY

                 DEVELOPED FOR NASA/AMES

                 NCC2-554

                 PERFORMANCE ANALYSIS OF LAN       ;

begin

external class level_2;
%INCLUDE prompt_user.sim
%INCLUDE read_input.sim
%INCLUDE get_tau.sim

integer num_of_stations;
integer num_typea_stations;
integer max_frame_size;
integer aver_msg_size;
long real prop_delay;
long real tau;
long real interrupt_time;
long real b_interrupt_time;
long real       data_rate;       ! data transmission rate,Mbp secs;
long real       dma_xfer_rate;   ! data transmission rate,Mbp secs;
long real       b_dma_xfer_rate;        ! data transmission rate,Mbp secs;
long real aver_arrival_time ;            ! mean time between message arrivals ;
(
! get the user input;
prompt_user;


! get the station parameters;
if ( num_typea_stations = num_of_stations) then
begin
        read_input("station_typea");
        b_interrupt_time := 0.0;
        b_dma_xfer_rate := 0.0;
end
else if ( num_typea_stations = 0) then
begin
        read_input("station_typeb");
        b_interrupt_time := interrupt_time;
        b_dma_xfer_rate := dma_xfer_rate;
        interrupt_time := 0.0;
        dma_xfer_rate := 0.0;
end
else
begin
        read_input("station_typeb");

        ! save the interrupt time and dma_xfer_rate;
        b_interrupt_time := interrupt_time;
        b_dma_xfer_rate := dma_xfer_rate;

        ! get type a input parameters;
        read_input("station_typea");
end;
```

```
tau := get_tau;

! start the simulation;
'  'el_2(num_of_stations,tau,prop_delay,interrupt_time,
        data_rate,dma_xfer_rate,aver_arrival_time,
        max_frame_size,aver_msg_size,b_interrupt_time,
        b_dma_xfer_rate,num_typea_stations)
begin
ref (message) cur_message;          ! test variable;
long real sim_start_time;                   ! starting time for simulation;
long real sim_end_time;             ! ending time for simulation;
integer i;

%INCLUDE print_data.sim

        ! assign the start time;
        sim_start_time := time;


                ! check for successful messages;
                while ( collectionq.total_message_success <
                        MIN_NUM_PER_STATION * num_of_stations ) do
                        begin
                                hold(10);
                        end;

                        ! assign the end time;
                        sim_end_time := time;

                        ! collect the data;
                        collectionq.get_station_data;

                        ! get the network throughput;
                        collectionq.get_network_data(sim_start_time,
                                sim_end_time);

                        ! print_data;
                        print_data;

end;
end
```

```
!**********************************************************************;
!;
!                integer procedure get_num_of_stations;

!        called by: network;
!;
!        returns the number of stations;
!;
!        globals used: none;
!;
!        actions: queries the user for number of stations;
!;
!;
!;
!;
!;
!;
!;
!**********************************************************************;
integer procedure get_num_of_stations;
begin
        ! get the number of stations;

        outtext(" Please input the number of stations");
        outimage;

        get_num_of_stations := inint;

end;
```

```
!                   SANTA CLARA UNIVERSITY

                    DEVELOPED FOR NASA/AMES

                    NCC2-554

                    PERFORMANCE ANALYSIS OF LAN      ;

!************************************************************


        procedure read_input


        called by:      network

        calls:          none

        action:         reads the data for stations
                        opens and closes the input file

        variables used: data_rate,
                        interrupt_time,
                        aver_msg_size,
                        max_frame_size,
                        aver_arrival_time
                        prop_delay
(

******************************************************************/;
procedure read_input(input_file);
text    input_file;
begin

ref(infile)     in_file;

        ! open the input file;
        in_file :- new infile(input_file);
        in_file.open(blanks(80));

        ! get station data;
        data_rate := in_file.inreal; in_file.inimage;
        interrupt_time := in_file.inreal; in_file.inimage;
        dma_xfer_rate := in_file.inreal; in_file.inimage;
        aver_arrival_time := in_file.inreal; in_file.inimage;
        prop_delay := in_file.inreal; in_file.inimage;
        max_frame_size := in_file.inint; in_file.inimage;
        aver_msg_size := in_file.inint; in_file.inimage;

        ! close the file;
        in_file.close;

end;
```

```
!***************************************************************;
!;
!               integer procedure prompt_user;
!
!       called by: network;
!;
!       returns the number of stations;
!;
!       globals used: none;
!;
!       actions: queries the user for number of stations;
!;
!;
!;
!;
!;
!;
!;
!***************************************************************;
procedure prompt_user;
begin
integer response;

        ! get the number of stations;

        outtext(" Please input the number of stations");
        outimage;
        outimage;

        num_of_stations := inint;

        outtext(" Please choose station configuration");
        outimage;
        outimage;
        outtext("                                       TYPE A            TYPE B");
        outimage;
        outimage;
        outtext("  interrupt time                       1.0               0.5");
        outimage;
        outtext("  dma transfer rate                    40.0              80.0");
        outimage;
        outtext("  maximum frame size                   1500              1500");
        outimage;
        outtext("  average interarrival time            0.000004          0.000004");
        outimage;
        outtext("  average message size                 5000              5000");
        outimage;
        outimage;

        outtext("PLEASE ENTER CHOICE");
        outimage;
        outtext("     TYPE A     1");
        outimage;
        outtext("     TYPE B     2");
        outimage;
        outtext("     MIXTURE    3");
        outimage;

        ! get the response;
        response := inint;

        ! assign the variables;
        if ( response = 1 ) then
        begin
                num_typea_stations := num_of_stations;
        end
        else if ( response = 2 ) then
        begin
                num_typea_stations := 0;
        end
        else
        begin
                ! wants a combination of types;
                ! get the number of type A stations;
                outtext(" Please input the number of TYPE A stations");
                outimage;
                outimage;

                num_typea_stations := inint;

                ! make sure we got an o.k. number;
                if ( num_typea_stations > num_of_stations ) then
                begin
                        num_typea_stations := num_of_stations;
                end
                else if ( num_typea_stations < 0 ) then
                begin
                        num_typea_stations :=  0;
                end;
        end;
 ( i;
!               SANTA CLARA UNIVERSITY

                DEVELOPED FOR NASA/AMES

                NCC2-554

                PERFORMANCE ANALYSIS OF LAN      ;

!************************************************************



        procedure print_data


        called by:      network

        calls:          none

        action:         prints the data for stations
                        and networks
                        opens and closes the output file

        variables used: collection_queue variables


*************************************************************/;
(  )cedure print_data;
begin

ref(outfile)    out_file;

        ! open the output file;
        out_file :- new outfile("output");
        out_file.open(blanks(80));

        ! print station data;
        out_file.outtext("station # ");
        out_file.outtext(" throughput            ");
        out_file.outtext("aver delay / frame ");
        out_file.outtext("success ");
        out_file.outtext("reject ");
        out_file.outimage;

        for i := 1 step 1 until num_of_stations do
        begin
                out_file.outint(i,5);
                out_file.outtext("  ");
                out_file.outreal(collectionq.throughput(i),10,20);
                out_file.outtext("  ");
                out_file.outreal(collectionq.aver_delay_per_frame(i),10,20);
                out_file.outtext("  ");
                out_file.outint(collectionq.message_success(i),5);
                out_file.outint(collectionq.message_not_sent(i),5);
                out_file.outimage;

        end;
        ! print the network characteristics;
        out_file.outtext(" network_throughput = ");
        out_file.outreal(collectionq.network_throughput,10,20);
        out_file.outtext(" aver_delay_per_frame = ");
        out_file.outreal(collectionq.network_aver_delay,10,20);
        out_file.outimage;
        out_file.outtext(" simulation time ");
        out_file.outreal((sim_end_time - sim_start_time),10,20);
        out_file.outimage;

        ! close the file;
        out_file.close;

end;
```

```
!*********************************************************************;
!;
!               long real procedure get_tau;
!
!       called by: network;
!;
!       returns tau;
!;
!       globals used: ;
!               long real max_net_length;
!               long real       prop_delay;
!;
!       actions: calculates tau;
!;
!;
!;
!*********************************************************************;
long real procedure get_tau;
begin

long real max_net_length;

        ! assign variables;
        max_net_length := 2.5;   ! max length of network;

        get_tau := max_net_length * prop_delay;

end;
```

```
!*************************************************************************

  HEAD CLASS collection_queue

        defines the queue used to collect data

        called by: level_2.sim - allocates collection queue
                   tx_tcp.sim    - puts messages into queue

        calls:          none

        actions:        defines collection queue
                        initializes collection queue variables

**************************************************************************;


  HEAD CLASS collection_queue;
  BEGIN

integer array total_num_of_bytes(1:num_of_stations);
                                ! total num_of data bytes;
integer array total_num_of_messages(1:num_of_stations);
                                ! total num_of data messages;
integer array total_num_of_frames(1:num_of_stations);
                                ! total num_of frames sent;
integer array message_success(1:num_of_stations);
                                ! number of messages successfully
                                        received;
integer array message_not_sent(1:num_of_stations);
                                ! number of messages not sent;
long real       array total_through_time(1:num_of_stations);
                                ! total time traveling through;
long real       array throughput(1:num_of_stations);
                                ! total # bytes / total time ;
long real       array aver_delay_per_frame(1:num_of_stations);
                                ! total time per message / total # frames ;
long real       network_throughput;
long real       network_aver_delay;
integer         total_message_success;
integer i;

%INCLUDE collect_data.sim
%INCLUDE get_station_data.sim
%INCLUDE get_network_data.sim

        ! main body;
        ! initialize variables;
        for i:= 1 step 1 until num_of_stations do
        begin
                total_num_of_bytes(i) := 0;
                total_num_of_messages(i) := 0;
                total_num_of_frames(i) := 0;
                message_success(i) := 0;
                total_through_time(i) := 0.0;
                throughput(i) := 0.0;
                aver_delay_per_frame(i):= 0.0;
        end;

        network_throughput := 0.0;
        total_message_success := 0;

    END;
```

```
!*****************************************************************;
!;
!                   integer procedure get_frames_per_message;

!         called by: collect_data;
!;
!         returns the number of frames per message;
!;
!         globals used: message.bytes;
!;
!         actions: calculates frames per message;
!;
!;
!;
!;
!;
!;
!;
!*****************************************************************;
integer procedure get_frames_per_message(num_of_bytes);
integer num_of_bytes;
begin
integer bytes_remaining;
integer num_of_frames;


        ! initialize variables;
        bytes_remaining := num_of_bytes;
        num_of_frames := 0;

        while( bytes_remaining > 0 ) do
        begin
                ! increment frame count;
                num_of_frames := num_of_frames + 1;

                ! decrement bytes_remaining;
                bytes_remaining := bytes_remaining -
                        BYTES_PER_FRAME;
        end;

        get_frames_per_message := num_of_frames;

end;
```

```
!***************************************************************;
!;
!                 procedure collect_data;

!        called by: collection_queue ;
!;
!         collects station data for messages;
!;
!        globals used: ;
!                 message.createtime;
!                 message.donetime;
!                 message.totaltime;
!                 message.rejected;
!                 dataunit.bytes;
!;
!        actions: collects data statistics for messages;
!                 sets msg.time_per_message;
!                 collects time per station;
!                 collects frames per station;
!                 collects the # of useful bytes ( data bytes );
!                         per station;
!;
!;
!;
!;
!;
!;
!;
(  ***************************************************************;
procedure collect_data(cur_message);
ref (message) cur_message;       ! current message examined;
begin
%INCLUDE         get_frames_per_message.sim
%INCLUDE         inc_message_count.sim

! did we get a message;
if  ( cur_message =/= none ) then
begin
        ! do we have a data message;
        ! if so, collect data;
        if ( cur_message.type = DATA ) then
        begin
                ! was this a successful transmission;
                if ( cur_message.donetime  <> 0.0 ) then
                begin
                        if ( not cur_message.rejected ) then
                        begin
                          ! collect data;

                          ! increment the message count;
                          inc_message_count(cur_message.source_addr);

                          ! get the time it took to send the message;
                          cur_message.time_per_message :=
                                        cur_message.donetime -
                                        cur_message.createtime;

                          ! add it to sending station time;
                          total_through_time(cur_message.source_addr) :=
                                total_through_time(cur_message.source_addr) +
                                        cur_message.time_per_message ;
                                        !   increment the number of messages;
                                        total_num_of_messages(cur_message.source_addr) :=
                                              total_num_of_messages(cur_message.source_addr)
                                                     + 1;

                                        ! collect the number of bytes;
                                        total_num_of_bytes(cur_message.source_addr) :=
                                              total_num_of_bytes(cur_message.source_addr) +
                                                     cur_message.bytes;

                                        ! get num_of_frames per message;
                                        total_num_of_frames(cur_message.source_addr):=
                                          total_num_of_frames(cur_message.source_addr) +

get_frames_per_message(cur_message.bytes);

                                     end;
                            end;
                  end;
                  ! take the message out of the queue;
                  cur_message.out;

end
else
begin
         outtext(" Error in get_data. msg is NULL pointer");
         outimage;
end;
(   l;
!*************************************************************;
!;
!               procedure inc_message_count;

!       called by: collect_data;
!;
!       returns none;
!;
!       globals used: message_success;
!;
!       actions: increments the number of successful messages
                per station. If the number of messages per station
                is less than MIN_NUM_PER_STATION, then the total
                number of messages per network is incremented.
                        For the simulation to be successful in
                data collection, every station must have a guaranteed
                minimun throughput.;
!;
!;
!;
!;
!;
!;
!;
!*************************************************************;
procedure inc_message_count(id);
integer id;
begin
(
        !increment the number of station messages;
        message_success(id) := message_success(id) + 1;

        ! should we increment the net success messages?;
        if ( message_success(id) <= MIN_NUM_PER_STATION  ) then
                total_message_success := total_message_success + 1;
end;
```

```
!*****************************************************************;
!;
!                  procedure get_station_data;
!
!        called by: network ( call to simulation );
!;
!         collects station data for messages;
!;
!        globals used: ;
!        total_time;
!        total_through_time;
!        total_num_of_bytes;
!        total_num_of_messages;
!        dataunit.bytes;
!;
!        actions: collects data statistics for messages;
!                 collects time per station;
!                 collects the # of useful bytes ( data bytes );
!                         per station;
!;
!;
!;
!;
!;
!;
!;
!*****************************************************************;
procedure get_station_data;
begin

integer i;

        ! loop through the stations;
        for i:= 1 step 1 until num_of_stations do
        begin
                !check for zero divide;
                if ( total_through_time(i)  <> 0.0 ) then
                begin
                        ! calculate the throughput for this station;
                        throughput(i) := total_num_of_bytes(i) /
                                 total_through_time(i)  ;

                        ! calculate average delay per frame for this station;
                        aver_delay_per_frame(i) := total_through_time(i) /
                                 total_num_of_frames(i)  ;
                end;
        end;
end;
```

```
!*************************************************************;
!;
!               procedure get_network_data;
!
!       called by: network ( call to simulation );
!;
!        collects station data for messages;
!;
!       globals used: ;
!               network_throughput;
!               calculates network time from sim_end - sim_start;
!;
!       actions: collects data statistics for network;
!               calculates throughput per network;
!               calculates average packet delay per network;
!;
!;
!;
!;
!;
!;
!;
!*************************************************************;
procedure get_network_data(sim_start_time,sim_end_time);
long real       sim_start_time;         ! starting time of simulation;
long real       sim_end_time;           ! ending time of simulation;
begin
        ! collection variables;
        integer net_total_num_of_bytes; ! total num_of data bytes;
        long real net_total_through_time;
        long real net_total_aver_delay;
        long real       throughput;
        integer i;

        ! initialize variables;
        net_total_through_time:=  sim_end_time - sim_start_time;
        net_total_num_of_bytes:= 0;
        throughput := 0.0;
        net_total_aver_delay := 0.0;

        ! loop through the station collection queue;
        for i := 1 step 1 until num_of_stations do
        begin
                ! collect the number of bytes;
                net_total_num_of_bytes := net_total_num_of_bytes +
                        total_num_of_bytes(i);

                ! collect the average delay per frame;
                net_total_aver_delay := net_total_aver_delay +
                        aver_delay_per_frame(i);
        end;

        ! calculate the throughput for this network;

        if (  net_total_through_time <> 0.0 ) then
        begin
                network_throughput := net_total_num_of_bytes /
                        net_total_through_time  ;
        end;

        ! calculate mean delay per frame;
        if (  net_total_aver_delay <> 0.0 ) then
```

```
          begin
                  network_aver_delay := net_total_aver_delay /
                          num_of_stations   ;
          end;

  end;
```

```
!                SANTA CLARA UNIVERSITY

                 DEVELOPED FOR NASA/AMES

                 NCC2-554

                 PERFORMANCE ANALYSIS OF LAN      ;


!-----------------------------------------------------------------------------;
!                                                                             ;
!    TX_HOST - transmitter host                                               ;
!                                                                             ;
!    STATES:   Not Applicable                                                 ;
!                                                                             ;
!    Actions:  Use station statistics to generate the arrival and lengths     ;
!              of new messages.  The messages are put into a host message     ;
!              queue and the corresponding TCP layer is notified (activated);
!              to allow action if its state/buffer permits.                   ;
!                                                                             ;
!    Globals Used:  U - the random seed number used as input to all random    ;
!                       number generations.                                   ;
!                   message_count - unique id numbers for the messages, for   ;
!                       tracking and debugging primarily.                     ;
!       History
        1/11/89         mhall           change random number generator
                                        from randint to negexp

;
(       1/19/89         mhall           put a check for
                                        MAX_MESSAGE_COUNT before
                                        creating message. keep track of
                                        rejected attempts when queue is
                                        full
        2/17/89         mhall           changed code to use
                                        aver_msg_size
;
!-----------------------------------------------------------------------------;
host CLASS tx_host;
BEGIN
long real arrival_time;                     ! time until arrival;
integer message_status;

        procedure generate_messages;
        begin
                ! generate arrival of a message            ;
                arrival_time := negexp( aver_arrival_time, u);

                hold(arrival_time); ! no action until message arrival time   ;

                ! can we generate another message;
                if ( msg_queue.cardinal < MAX_HOST_MESSAGE_COUNT ) then
                begin
                        ! create and initialize the new message      ;
                        host_msg :- new message;
                        host_msg.createtime:= TIME;

                        host_msg.id := msg_count;

                        ! this parameter is used primarily to allow ;
                        ! for tracking of messages, a unique id#     ;
```

```
                    ! assign source and destination address;
                    host_msg.source_addr := id;

                    ! get a destination other than itself;
                    host_msg.dest_addr := randint( 1, num_of_stations, u);
                    while host_msg.dest_addr = id do
                    begin
                            host_msg.dest_addr :=
                                    randint( 1, num_of_stations, u);
                    end;

                    host_msg.type := DATA;

                    ! get a size for the message;
                    ! host_msg.bytes := randint( minbytes, maxbytes, u);
                    host_msg.bytes := aver_msg_size;

                    ! into the message queue;
                    host_msg.Into(msg_queue);
            end
            else
            begin
                    ! keep track of the ones that don't get in;
                    collectionq.message_not_sent(id) :=
                            collectionq.message_not_sent(id) + 1;
            end;
        end;
(
begin                           ! this layer doesn't passivate on initialization;
        msg_queue :- new message_queue("tx_host");


        ! set characteristics of station message;
        !if (id <= (num_of_stations/2)) then
        begin
                minbytes := 50;
        !       maxbytes := 150;
        !end else begin
                minbytes := 8000;
        !       maxbytes := 10000;
        !end;

        while TRUE do           ! generate messages forever ;
        begin

                ! generate messages;
                generate_messages;
                if stx_tcp(id).IDLE then
                begin
                        activate stx_tcp(id);
                end;
                hold(interrupt_time);

        end--of--while;
end;
END;
```

```
!                   SANTA CLARA UNIVERSITY

                    DEVELOPED FOR NASA/AMES

                    NCC2-554

                    PERFORMANCE ANALYSIS OF LAN      ;
!--------------------------------------------------------------------------;
!                                                                          ;
!    RX_HOST - receiver host                                               ;
!                                                                          ;
!    STATES:   FREE, RECEIVING                                             ;
!                                                                          ;
!    Actions:  Receive a completed message from the TCP layer.  Delay to  ;
!              simulate the DMA transfer of data.  Reset to be ready to   ;
!              receive the next message.                                   ;
!                                                                          ;
!--------------------------------------------------------------------------;
host CLASS rx_host;
BEGIN

procedure dma_transfer(no_bytes);
!------------------------------------------------------------;
! procedure dma_transfer                                     ;
!                                                            ;
! This procedure executes a hold to simulate a dma transfer.;
! It used the number of bytes passed to it to determine the ;
! actual lenght of the hold.                                 ;
(                                                            ;
! Globals: dma_xfer_rate                                     ;
!------------------------------------------------------------;
integer no_bytes;
begin
    hold(no_bytes * 8 /dma_xfer_rate);
end;


        boolean procedure receive(recvmsg);
        REF(message) recvmsg;
        begin
                recvmsg.INTO(msg_queue);
                dma_transfer(recvmsg.bytes);
                receive := TRUE;
        end;



        begin               ! RX_HOST MAIN ;
                ! initialize the rx_host entity                ;
                msg_queue :- new message_queue("rx_host");
                host_msg :- new message;

                ! passivate after creation;
                passivate;

                while TRUE do
                ! loop until the simulation ends               ;
                begin
                        if (msg_queue.EMPTY) then
                        ! no action required, do nothing       ;
```

```
                                        passivate;
                                end
                                else
                                begin
                                ! message queue not empty, begin processing ;
                                        state := RECEIVING;
                                        ! receive a message from TCP layer     ;
                                        host_msg :- msg_queue.SUC;

                                        if (host_msg =/= NONE) then
                                        ! NONE test to avoid runtime errors ;
                                        begin
                                                ! simulate processing time          ;
                                                hold(interrupt_time);

                                                state := FREE;

                                                ! remove the message from the queue  ;
                                                host_msg.OUT;

                                                ! discard the message               ;
                                                host_msg :- NONE;
                                end;
                        end;
                      end;
            end;
END++of++rx_host;
```

```
!                   SANTA CLARA UNIVERSITY

                    DEVELOPED FOR NASA/AMES

                    NCC2-554

                    PERFORMANCE ANALYSIS OF LAN     ;

! TX_TCP.SIM - used as include file in TCP.SIM   ;

!-------------------------------------------------------------------------;
!                                                                         ;
!    TX_TCP -  transmitter tcp                                            ;
!                                                                         ;
!    STATES:   FREE, SENDING                                              ;
!                                                                         ;
!    Actions:  Fetch message from host message queue, packetize,          ;
!              add to buffer, send the message.                           ;
!              Send acknowledgments, and piggybacked data/ack frames      ;
!-------------------------------------------------------------------------;

tcp CLASS tx_tcp;
BEGIN
REF(buffer) ackq;
REF(message_queue) host_msg_queue;
INTEGER rc;
BOOLEAN ack_to_send, piggyback;                   ! Flags used to determine the;
                                                  ! next action within the main;
                                                  ! loop of the tcp transmitter;



procedure dma_transfer(no_bytes);
!--------------------------------------------------------------;
! procedure dma_transfer                                        ;
!                                                               ;
! This procedure executes a hold to simulate a dma transfer.;
! It used the number of bytes passed to it to determine the ;
! actual lenght of the hold.                                    ;
!                                                               ;
! Globals: dma_xfer_rate                                        ;
!--------------------------------------------------------------;
integer no_bytes;
begin
    hold(no_bytes * 8 /dma_xfer_rate);
end;

    procedure packetize(msg);
    REF(message) msg;
    !----------------------------------------------------------------------;
    !  This procedure divides the message referenced by MSG into frames.   ;
    !  The frames are put into the tcp buffer by the netservice level      ;
    !  procedure RECEIVE.  The final frame is marked by setting the FIN bit. ;
    !  This procedure is called only after space within the buffer has been ;
    !  reserved for the entire message size.  A partial message will not be ;
    !  packetized.                                                         ;
    !----------------------------------------------------------------------;
    begin
    REF(frameunit) tcpframe;
    integer temp, msg_size;
       i := 1;                                   ! counter for number of frames   ;
    begin                                          ! divide the message into frames ;
        temp := msg.bytes;
        msg_size := max_frame_size;
        while temp > 0 do
        begin                                      ! copy info from message to frame;
            tcpframe :- new frameunit;
            tcpframe.id := msg.id;
            tcpframe.bytes := msg_size;
            tcpframe.dest_addr := msg.dest_addr;
            tcpframe.source_addr := msg.source_addr;
            tcpframe.type := DATA;
            tcpframe.seqnum := msg_size * i;
            tcpframe.msgptr :- msg;
            if tcpframe.seqnum >= msg.bytes then
            begin                                  ! last frame in the message      ;
                tcpframe.fin := True;              ! set the Finished bit           ;
                tcpframe.bytes :=temp;
                tcpframe.seqnum := msg.bytes;
                if (tcpframe.bytes < 65) then
                begin                              ! minimum frame size is 65 bytes;
                                                   ! adjust bytes & seqnum          ;
                    tcpframe.bytes := 65;
                    tcpframe.seqnum  := msg.bytes + (65 - temp);
                end;
                buf.msgtail:-tcpframe;             ! set pointer to end of message ;
            end;
            temp := temp - msg_size;               ! calculate bytes left in message;
            return_code := receive(tcpframe);      ! call routine to put frame;
                                                   ! into transmittor buffer  ;
            i := i + 1;                            ! increment frame counter        ;
        end;
    end;
end--of--packetize;


procedure ack_message(ackframe);
REF(frameunit) ackframe;
!-------------------------------------------------------------------------------;
!       ack_message(ackframe)                                                   ;
!       processes the receipt of an acknowledgement                             ;
!        - updates the window                                                   ;
!        - removes frames from buffer when last frame acknowledged              ;
!        - sets donetime of message                                             ;
!    assumptions:                                                               ;
!     - one message in the buffer at a given time                               ;
!-------------------------------------------------------------------------------;
begin
    REF(frameunit) tempframe;
    REF(message) msgptr;

    if win.state <> EMPTY then      ! if the window is empty, any ack that   ;
                                    ! received will be ignored               ;
     begin
        tempframe :- win.front;     ! starting at the front of the window,   ;
                                    ! search the frames in the window for    ;
                                    ! frames that have a seqnum less than    ;
                                    ! or equal to the acknum, remove those   ;
                                    ! frames from the window.                ;
        while tempframe=/=NONE
            and tempframe.dest_addr=ackframe.source_addr
            and tempframe.seqnum<=ackframe.acknum do
            kill_timer(tempframe); ! cancel the timer for the acked frame;
                                   ! adjust the window size               ;
            win.cursize := win.cursize - tempframe.bytes;
            if tempframe.fin then  ! when the last frame in message       ;
            begin                  ! remove the message from buffer       ;
                msgptr :- tempframe.msgptr;
                win.front :- tempframe.SUC;
                if (outofbuf(tempframe.id) = OK) then
                begin              ! Mark the message with current time   ;
                                   ! indicating the time the message      ;
                                   ! acknowledgement was received.  Call  ;

                                   ! for data collection.                 ;
                    if (set_donetime(msgptr) = OK) then
                    begin
                            ! get the data from this successful message;
                            collectionq.collect_data(msgptr);

                    end;
                end;
                if win.front==NONE then
                begin
                    win.reset_window;
                    goto Break;
                end else begin
                    win.state := NENF;
                    tempframe :- win.front;
                    goto Break;
                end;
            end else begin
                win.front :- tempframe.SUC;
                if win.front==NONE then
                begin
                    win.reset_window;
                    goto Break;
                end;
                tempframe :- win.front;
            end;
        end;
        Break:          ! break out of while loop, incase pointer is NULL;
    end;
end--of--ack;


procedure copyframe(source, dest);
REF(frameunit) source, dest;
!-------------------------------------------------------------------------;
!  This procedure makes a copy of the frame being sent, since SIMULA      ;
!  will not allow a given Link class object to exist in more than one     ;
!  set (ie. a frame cannot be in more than one buffer at any given time).;
!  To allow the frame to stay in the transmittor's buffer until it is     ;
!  acknowledged, a copy of the frame must be sent.                        ;
!-------------------------------------------------------------------------;
begin
    dest.id := source.id;
    dest.bytes := source.bytes;
    dest.dest_addr := source.dest_addr;
    dest.source_addr := source.source_addr;
    dest.type := source.type;
    dest.seqnum := source.seqnum;
    dest.acknum := source.acknum;
```

```
        dest.fin := source.fin;
        dest.msgptr :- source.msgptr;
    end;

    boolean procedure transmit(frame);
    REF(frameunit) frame;
    !-----------------------------------------------------------------------------;
    !  This procedure copies the frame to be sent and calls the IP RECEIVE      ;
    !  procedure.  If IP receives the frame, activate the IP entity.            ;
    !                                                                           ;
    !  Returns:  TRUE - frame received by IP                                    ;
    !            FALSE - frame not received by IP                               ;
    !-----------------------------------------------------------------------------;
    begin
    REF(frameunit) sendframe;
        sendframe :- new frameunit;   ! create new frame_unit template to send;
        copyframe(frame, sendframe);  ! copy info from old frame to new copy ;
        if ( stx_ip(id).receive(sendframe) ) then
        begin
            transmit := TRUE;         ! frame received by IP, its on its way ;
            activate stx_ip(id);      ! IP has something to do, so wake it up;
        end else  begin
            transmit := FALSE;        ! the frame couldn't be sent yet       ;
        end;
    end--of--transmit;


    REF(timeout_unit) procedure find_timeout;
    !-----------------------------------------------------------------------------;
    !  This procedure searches the timeout queue to find the frame that         ;
    !  timed out, ie) the frame in which the FRAME_TIMED_OUT flag is set.       ;
    !                                                                           ;
    !  Returns: reference to the first timeout_unit found which meets the       ;
    !           condition FRAME_TIMED_OUT, otherwise NONE.                      ;
    !-----------------------------------------------------------------------------;
    begin
    REF(timeout_unit) current, next;
    boolean found;
        find_timeout :- NONE;
        found := FALSE;               ! flag to indicate if timeout was found ;
        current :- timeoutq.FIRST;    ! start at the beginning of the timeoutq;
        while (not found) do
        begin
            if  (current =/= NONE) then      ! search until end of queue      ;
            begin
                if (current.status = FRAME_TIMED_OUT) then
                begin                 ! found a timed out frame               ;
                    find_timeout :- current;
                    found := TRUE;
                    timeout_count := timeout_count -1;
                end else begin        ! didn't find one, get the next timeout ;
                    current :- current.SUC;
                end;
            end else begin                ! have searched the entire queue    ;
                found := TRUE;            ! break out of the loop             ;
                timed_out := FALSE;
                timeout_count := timeout_count -1;
            end;
        end;
        if (timeout_count = 0) then timed_out := FALSE;
    end;
```

```
procedure update_buffer;
!-------------------------------------------------------------------------;
!  This procedure updates the necessary buffer pointers after a frame     ;
!  is transmitted.                                                        ;
!-------------------------------------------------------------------------;
begin
    lastsent :- frame;
    buf.current :- (buf.current).SUC;
end--update_buffer;

boolean procedure receive_ack(ackframe);
REF(frameunit) ackframe;
!-------------------------------------------------------------------------;
!  This procedure receives an acknowledge frame for transmission from     ;
!  the TCP Receiver.  Comparisons are made to determine:                  ;
!             - if the frame is to be piggybacked                         ;
!             - if an previous ACK for the same message exists and        ;
!                  should be updated to reflect the current status of     ;
!                  the message                                            ;
!             - if this ACK should be added to the acknowledgement        ;
!                  queue                                                  ;
!                                                                         ;
!  Returns: TRUE if the acknowledgment is successfully handled            ;
!           FALSE if there is an error in the processing                  ;
!-------------------------------------------------------------------------;
begin
REF(frameunit) tempframe;
boolean set_piggyback;              ! flag to indicate if piggyback       ;
                                    ! option was used                     ;
    set_piggyback := FALSE;        ! initialize flag to indicate not used;
    if (lastsent =/= NONE) then     ! NONE test to avoid runtime error    ;
    begin
        tempframe :- lastsent.SUC;  ! get the next packet to be sent      ;
    end else begin
        tempframe :- NONE;
    end;
                          !check first packet for matching destination    ;
                          !address - if match found then piggyback the ack ;
    if (tempframe =/= NONE) then
    begin
        if tempframe.dest_addr = ackframe.dest_addr then
        begin
            piggyback := TRUE;      ! set flag to show ack piggybacked    ;
            set_piggyback := TRUE;
            tempframe.ack := TRUE;  ! mark frame with acknowledgement info;
            tempframe.acknum := ackframe.acknum;
        end;
    end;
    if (not set_piggyback) then     ! if ack not already piggybacked      ;
    begin
        if (ackq.EMPTY) then
        begin                       ! no ack exist in the ACKQ            ;
            ackframe.INTO(ackq);    ! put the new ack into the ack queue  ;
        end else begin              ! otherwise check the acks in the     ;
                                    ! queue to see if updating an         ;
                                    ! existing ack is appropriate         ;
            tempframe :- ackq.SUC;  ! get the first entry in the ACKQ     ;
            while (tempframe =/= NONE) and (ackframe=/= NONE) do
            begin
                if (tempframe.id=ackframe.id) and
                   (tempframe.acknum < ackframe.acknum) then
```

```
                    tempframe.acknum := ackframe.acknum;
                    ackframe :- NONE;   ! ack frames combined, discard    ;
                                        ! ack not used                    ;
                end else begin      ! match not found, look at next ack  ;
                    tempframe :- tempframe.SUC;
                end;
            end;
            if (ackframe=/=NONE) then    ! no matching ACK found in ACKQ ;
            begin
                ackframe.INTO(ackq);    ! put ack into the ack queue      ;
            end;
        end;
    end;
    ack_to_send := TRUE;
    receive_ack := TRUE;
end--of--receive_ack;


procedure transmit_failed(frameptr);
REF(frameunit) frameptr;
!------------------------------------------------------------------------;
! This procedure removes the entire message from the buffer and window   ;
! when transmission at the csmacd layer has failed.                      ;
!                                                                        ;
!  Assumptions:                                                          ;
!     - only one message in the tranmission window at any given time     ;
!------------------------------------------------------------------------;
begin
    (frameptr.msgptr).rejected := TRUE;   ! mark the message as rejected  ;
                                          ! this info will be used later  ;
                                          ! in the network statistics     ;
    return_code := outofbuf(frameptr.id);
    win.reset_window;
end;

REF (timeout_unit)      procedure create_timeout_ref(timeout_frame);
REF(frameunit) timeout_frame;
!------------------------------------------------------------------------;
!  This procedure creates a timeout_unit for placement into the timeout  ;
!  queue.  It initializes the parameters of the timeout_unit.            ;
!                                                                        ;
!  Returns: Reference to a new timeout_frame                             ;
!                                                                        ;
!  NOTE: TCP actually calculates the time_up value using a weighted      ;
!        average of the actual times to send and receive an ACK.         ;
!        This feature has not been implemented in this version.          ;
!------------------------------------------------------------------------;
begin
    REF(timeout_unit) timeout;
    timeout :- new timeout_unit;      ! create a new timeout template    ;
                                      ! initialize all of the variables  ;
    timeout.layer :- stx_tcp(id);     ! set reference to the entity      ;
    timeout.frameptr :- timeout_frame; ! set reference to specific frame ;
    timeout.time_up := 28000.0;
    create_timeout_ref :- timeout;
end;

procedure kill_timer(timeout_frame);
REF(frameunit) timeout_frame;
!------------------------------------------------------------------------;
!  This procedure searches the timeout queue for the reference to the    ;
!  acknowledged frame.  If found it is cancelled and removed from the    ;
!   timeout q.  If it is not found no action occurs. The search ends     ;
!   with the first match found. There is only one timeout_frame for any  ;
!   frame transmitted, including any re-sends of the same frame          ;
!   for whatever reason.                                                 ;
!-------------------------------------------------------------------------;
begin
    REF(timeout_unit) current;      ! pointer to current position in queue;
    current :- timeoutq.FIRST;      ! start at the front of the timeoutq   ;
    while (current=/=NONE) do       ! search until end of timeoutq found   ;
    begin
        if (current.frameptr==timeout_frame) then
        begin                       ! found timeout_unit for ref frame     ;
            cancel(current.timerptr);  ! cancel the timeout process        ;
                                    ! using the REF in the timeout_unit    ;
            current.OUT;            ! remove timeout_unit from queue       ;
            goto Break;             ! exit the loop                        ;
        end;
        current :- current.SUC;     ! frame match not found, get next unit;
    end;
Break:
end;

procedure start_timer(timeout);
REF(timeout_unit) timeout;
!-------------------------------------------------------------------------;
!  This procedure creates a new process TCP_TIMER to be an independent    ;
!  process for the tracking of a timeout.  If a timeout occurs the        ;
!  independent process will set the necessary flags using the REF's       ;
!  that are contained in the timeout_unit.  No references to the          ;
!  TCP_TIMER process exist, so that it will be discarded for garbage      ;
!  collection after it performs its given timeout function.               ;
!-------------------------------------------------------------------------;
begin
    REF(tcp_timer) new_tcp_timer;
    new_tcp_timer :- new tcp_timer; ! create a new timer process          ;
    timeout.timerptr :- new_tcp_timer;
    new_tcp_timer.setup(timeout);    ! initialize the timer process       ;
    timeout.INTO(timeoutq);          ! put in q (keep a reference to it)  ;
    activate new_tcp_timer;          ! start the timer                    ;
end;

begin              ! main program portion of tx_tcp;
REF(frameunit) ackframe;             ! reference to ack frame used in MAIN;
REF(message) msg;                    ! reference to a message used in MAIN;
REF(timeout_unit) timeoutptr;        ! reference to a timeout used in MAIN;
boolean xmit_failed;                 ! parameter used to determine the   ;
                                     ! status of attempted transmission  ;
                                     ! if TRUE, the layer passivates     ;
                                     ! waiting for a state change in an  ;
                                     ! adjacent layer to retry.          ;
xmit_failed := FALSE;
ackq :- new buffer;
msg :- new message;
buf.cursize := 0;                    ! Initialize the parameters         ;
buf.maxsize := 20480;                ! set tcp buffer size for station   ;
buf_space := buf.maxsize;
msg :- none;
lastsent :- none;
piggyback := FALSE;                  ! initialize loop flags            ;
ack_to_send := FALSE;
```

```
passivate;

! assign the host message queue to local pointer;
host_msg_queue :- stx_host(id).msg_queue;

while TRUE do                            ! DO FOREVER LOOP                    ;
begin
  while (host_msg_queue.EMPTY OR buf.state = FULL)
            and not ( ack_to_send OR timed_out OR next_msg ) do
  begin                               ! Nothing to do - passivate            ;
    passivate;
  end;
  if ( ack_to_send OR timed_out OR next_msg ) then
  begin                               ! perform activity associated with    ;
                                      ! flags in priority order             ;
    state := SENDING;
    if (ack_to_send ) then
    begin
        if (piggyback) then           ! First Priority - send piggyback ack   ;
        begin
            piggyback := FALSE;     ! Reset piggyback flag                    ;
            frame :- buf.current;
            if (frame =/= NONE) then  ! NONE test to avoid runtime error ;
            begin
              if (win.reserve_space(frame.bytes)) then
              begin
                if ( transmit(frame)) then
                begin                 ! Frame transmitted, update win & buf   ;
                    if (win.addto(frame)) then
                    begin             ! Create timeout timer and start it     ;
                        timeoutptr :- create_timeout_ref(frame);
                        start_timer(timeoutptr);
                        update_buffer;
                    end else begin    ! set flag to passivate at end of loop;
                        xmit_failed := TRUE;
                    end--addto;
                end else begin
                    xmit_failed := TRUE;
                end--transmit;
              end else begin
                xmit_failed := TRUE;
              end--reserve;
            end;
        end else begin
            ackframe :- ackq.FIRST;      ! assign a reference to first ack;
            if (ackframe =/= NONE) then ! NONE test to aviod runtime error;
            begin
                if ( transmit(ackframe)) then  ! send the acknowledgement ;
                begin
                    ackframe.OUT;          ! remove the ack from the queue   ;
                                           ! if no more to send set flag     ;
                                           ! ack_to_send to FALSE            ;
                    if (ackq.EMPTY) then ack_to_send := FALSE;
                end else begin
                    xmit_failed := TRUE;! Transmit returned FALSE, set the;
                                           ! the xmit_failed flag so that the;
                                           ! transmitter will passivate at   ;
                                           ! end of this loop.  It will be   ;
                                           ! activated by IP or HOST when    ;
                                           ! either one's state changes.     ;
                end;
```

```
             if (ackq.EMPTY) then     ! verify there are no acks in queue;
             begin                    ! reset the ack_to_send flag       ;
                 ack_to_send := FALSE;
             end;
         end;
     end;
 end else begin                ! ack_to_send is FALSE, check other flags  ;
     if (timed_out) then
     begin
         timeoutptr :- find_timeout; ! find the timed out unit from q    ;
         if (timeoutptr =/= NONE) then
         begin                ! timeout unit found, resend by the frameptr;
             if (transmit(timeoutptr.frameptr)) then
             begin
                 timeoutptr.status := SET;  !start a new timer for frame;
                 start_timer(timeoutptr);
                 timeout_count := timeout_count - 1;
             end else begin
                 xmit_failed := TRUE;  ! set flag to cause passivate    ;
             end;
         end else begin
             timed_out := FALSE;  ! No timeout found, reset the flag    ;
         end;
     end else begin
       if (next_msg) then
       begin
         frame :- buf.current;
         if (frame =/= NONE) then
         begin
             if (win.reserve_space(frame.bytes)) then
             begin                      ! reserve space in the xmit window   ;
                 if (transmit(frame)) then
                 begin                  ! transmit the frame to IP            ;
                     if (win.addto(frame)) then
                     begin              ! officially add the frame to the win;
                                        ! create/start the timer              ;
                                        ! update the buffer pointers          ;
                         timeoutptr :- create_timeout_ref(frame);
                         start_timer(timeoutptr);
                         update_buffer;
                         if lastsent==buf.msgtail then
                         begin    ! send only one message at a time    ;
                             next_msg := FALSE;
                         end;
                     end else begin
                         xmit_failed := TRUE;
                         win.cancel_reserve_space(frame.bytes);
                     end;
                 end else begin
                     xmit_failed := TRUE;
                     win.cancel_reserve_space(frame.bytes);
                 end;
             end else begin
                 xmit_failed := TRUE;
             end;
         end else begin
             next_msg := FALSE;
         end;
      end;
  end;
end;
```

```
    if (xmit_failed) then       ! Xmit_failed flag indicates that some      ;
                                ! condition exists to attempt a transmit, and ;
                                ! that the attempt failed.  The transmitter   ;
                                ! will passivate, waiting for a change in     ;
                                ! the conditions, so that the next transmit   ;
                                ! might succeed.  The loop will be executed   ;
                                ! from the beginning after TX_TCP is activated;
                                ! so that whatever is highest priority will be;
                                ! done first.                                 ;
    begin
      xmit_failed := FALSE;
      passivate;
    end;
  end else begin
      if (not host_msg_queue.EMPTY) and (not buf.state=FULL) then
      begin
          state := RECEIVING;
          msg :- host_msg_queue.SUC;
                                 ! add message to tcp buffer only if         ;
                                 ! there is room for the entire message      ;
                                 ! buf_space indicates space remaining       ;
          if (reserve_buffer_space(msg.bytes)) then
          begin
              msg.Out;           ! Message is taken out of host queue and    ;
                                 ! added to the data collection queue.       ;


              msg.INTO(collectionq);
              dma_transfer(msg.bytes);     ! Hold for dma xfer of data bytes;
              packetize(msg);    ! Prepare the msg for sending, add to buffer ;
                                 ! as individual frames to be transmitted.    ;

          end;
          state := FREE;
      end;
  end;
  hold(interrupt_time);         ! Hold for tcp processing time,needs refinement;
  end--of--while;

end--of--main;
END++of++TCP;
```

```
!                SANTA CLARA UNIVERSITY

                 DEVELOPED FOR NASA/AMES

                 NCC2-554

                 PERFORMANCE ANALYSIS OF LAN      ;

! RX_TCP.SIM - used as include file in TCP.SIM   ;


!-----------------------------------------------------------------------------;
!                                                                             ;
!    RX_TCP -  receiver tcp                                                   ;
!                                                                             ;
!    STATES:   FREE, RECEIVING                                                ;
!                                                                             ;
!    Actions:  receive message from ip, update the buffer,                    ;
!              send ACK in response to data message, update parameters in     ;
!              response to CTRL, pass complete message up to host             ;
!     NOTE: For this simulation it is assumed that there is room in the       ;
!           receiver BUFFER for any arriving frame.  The WINDOW size will     ;
!           be checked for availability.                                      ;
!-----------------------------------------------------------------------------;
tcp CLASS rx_tcp;
BEGIN
REF(message) msg;
R̄ĒF(frameunit) summaryptr, temp;
{  EGER buf_rc;

procedure dma_transfer(no_bytes);
!---------------------------------------------------------------;
! procedure dma_transfer                                        ;
!                                                               ;
! This procedure executes a hold to simulate a dma transfer.;
! It used the number of bytes passed to it to determine the ;
! actual lenght of the hold.                                    ;
!                                                               ;
! Globals: dma_xfer_rate                                        ;
!---------------------------------------------------------------;
integer no_bytes;
begin
    hold(no_bytes * 8 /dma_xfer_rate);
end;

    integer procedure rx_receive(recvframe);
    REF(frameunit) recvframe;
    !-----------------------------------------------------------------;
    !  This procedure receives a frame at the tcp level.  It updates   ;
    !  window and buffer sizes to reflect the size of the frame        ;
    !  received.                                                       ;
    !-----------------------------------------------------------------;
    begin
    boolean win_rc;
        win_rc := FALSE;
        return_code := FAILED;

        if (win.maxsize < (win.cursize + recvframe.bytes)) then
        begin        ! set win.maxsize so never fails for WINDOW too small;
            win.maxsize := win.cursize + recvframe.bytes;
    if (win.reserve_space(recvframe.bytes)) then
    begin        ! reserve space in the window for the frame        ;
        if (buf.maxsize < (buf.cursize + recvframe.bytes)) then
        begin    ! set buf.maxsize so never fails for BUFFER too small;
            buf.maxsize := buf.cursize + recvframe.bytes +1;
        end;
        win_rc := win.addto(recvframe);
        buf_rc := receive(recvframe);
    end else begin
        win.cancel_reserve_space(recvframe.bytes);
    end;
    if (win_rc ) and (buf_rc = OK) then
    begin
        return_code := OK;
    end;
    if (buf.current == NONE) then
    begin
        buf.current :- recvframe;
    end;
    rx_receive := return_code;
end--of--rx-receive;

procedure ctrl_message;
begin
!-------------------------------------------------------------------;
!  This procedure can be used as a starting point to implement the  ;
!  control messages of TCP.                                         ;
!-------------------------------------------------------------------;
end--of--ctrl;

procedure reassemble_message;
!-------------------------------------------------------------------;
!  This procedure creates a new message and copies all of the       ;
!  information from the summary frame to the message preparing to   ;
!   pass the message up to the host.                                ;
!-------------------------------------------------------------------;
begin
    msg :- new message;       ! create a new message and copy info   ;
    msg.id := summaryptr.id; ! from the summary frame                ;
    msg.dest_addr := summaryptr.dest_addr;
    msg.source_addr := summaryptr.source_addr;
    msg.bytes := summaryptr.bytes;
end--of--reassemble_message;


ref(frameunit) procedure get_summary_frame;
!----------------------------------------------------------------------;
!   This procedure returns a frame pointer to the first frame          ;
!   found with the same message id number (id).  Since the             ;
!   buffer is FIFO, the first frame reference will be the              ;
!   considered the summary frame and will be updated with the          ;
!   receipt of additional frames of the same message to reflect        ;
!   the status of the message.  Non-contiguous frame will              ;
!   exist as separate units within the buffer.                         ;
!----------------------------------------------------------------------;
begin
    REF(frameunit) bufptr;
    bufptr :- buf.FIRST;                  ! start at the beginning of buf;
    while (bufptr =/= NONE) do            ! search to end of the buffer  ;
    begin
        if (frame.id=bufptr.id) then      ! matching frame found, assign ;
```

```
            get_summary_frame :- bufptr;
            go to Break;
        end else begin
            bufptr :- bufptr.SUC;          ! no match-check next buf entry;
        end;
    end;
    Break:
end--get_summary_frame;

integer procedure last_contiguous_byte;
!-----------------------------------------------------------------------;
!  This procedure will search through the buffer updating the summary   ;
!  frame until no more updates are possible.  The buffer pointers are   ;
!  updated and combined frames are removed from the buffer.             ;
!                                                                       ;
!  Returns:                                                             ;
!           Integer value of the highest contiguous byte received       ;
!-----------------------------------------------------------------------;
begin
    integer last_byte;
    boolean updated;
    REF(frameunit) bufptr,  tempptr;

    last_byte := 0;    ! set last_byte in case first frame not received ;
    updated := TRUE;
    while (updated) do
    begin ! search the list again after each update, until no more updates;
        bufptr :- summaryptr.SUC; ! start search after the summary frame   ;
        updated := FALSE;
        while (bufptr =/= NONE) do ! search until the end of the buffer    ;
        begin
            if (summaryptr.id=bufptr.id) then
            begin                     ! the frames have the same id number    ;
              if (summaryptr =/= bufptr) then
              begin                   ! the pointers point to different frames;
                if (bufptr.seqnum= (summaryptr.bytes+ bufptr.bytes)) then
                begin                 ! contiguous frame found, combine them   ;
                    summaryptr.seqnum := bufptr.seqnum;
                    summaryptr.bytes := summaryptr.bytes+bufptr.bytes;
                    last_byte := summaryptr.bytes;
                    summaryptr.fin := bufptr.fin;
                    updated := TRUE;
                end;
              end;
            end;
            if (updated) then
            begin      ! update the buffer pointers and remove the frame   ;
                       ! that was just combined with the summary frame     ;
                tempptr :- bufptr;
                bufptr :- bufptr.SUC;
                if (tempptr==buf.current) then
                begin
                    buf.current:-(buf.current).PREV;
                end;
                tempptr.OUT;
                tempptr:- NONE;
            end else begin
                bufptr :- bufptr.SUC;
            end;
        end--while;
    end--while;
```

```
     begin
         last_byte := summaryptr.bytes;
     end;
     last_contiguous_byte := last_byte;
end--of--last-contiguous-byte;

procedure acknowledge(aframe);
REF(frameunit) aframe;
!-----------------------------------------------------------------------------;
!  This procedure creates an acknowlegement for the last contiguous          ;
!  byte of the same message as the frame received.  The ACK is put           ;
!  into the acknowledgement queue of the transmitter.                         ;
!-----------------------------------------------------------------------------;
begin
ref(frameunit) ackframe;
    ackframe :- new frameunit; ! create new frame for the ack                ;
    ackframe.dest_addr := aframe.source_addr; ! initialize the frame         ;
    ackframe.source_addr := aframe.dest_addr;
    ackframe.id := aframe.id;
    ackframe.ack := TRUE;
    ackframe.type := ACK;
    ackframe.bytes:= 65;         ! set ack bytes to minimum packet size ;
    ackframe.acknum := last_contiguous_byte; ! find the byte to ack          ;
    ackframe.setwindow := buf.maxsize - buf.cursize;
    if not ( stx_tcp(id).receive_ack(ackframe)) then
    begin
    end;
    activate stx_tcp(id);
end--of--acknowledge;


!******** RX_TCP --- MAIN ****************************;
begin
    buf.cursize := 0;
    buf.maxsize := 10240;         ! set tcp buffer size for station;
                                  ! info set up in a file?;

    passivate;

    while TRUE do
    begin
      if buf.current =/= NONE then ! frame to receive in buffer, start ;
      begin                        ! the receive process               ;
            state := RECEIVING;
            frame :- buf.current;
                                   ! control not implemented  - 12/88   ;
            if frame.type = CTRL then ctrl_message;
                                   ! pass ack info to tx_tcp for updates;
            if frame.ack then stx_tcp(id).ack_message(frame);
                                   ! receive data information           ;
            if frame.type = DATA then
            begin
                                   ! Find the summary frame for message,;
                                   ! should return a pointer to the 1st ;
                                   ! frame found for the same message.  ;
                                   ! Either a previously received frame ;
                                   ! or the current frame if no previous;
                                   ! frame exists should be returned.   ;
                summaryptr :- get_summary_frame;
                if (summaryptr==NONE) then ! debug message              ;
                begin
                acknowledge(frame);
            end;
            ! remove frame from receiver window ;
            return_code := win.rxtcp_outof(frame);

            if (buf.current=/=NONE) then
                buf.current :- (buf.current).SUC;

            if (summaryptr =/= NONE) then
            begin
            if (summaryptr.fin) and
                    (summaryptr.bytes=summaryptr.seqnum) then
            begin
                ! last frame, message complete      ;
                reassemble_message;

                !send message to the host    ;
                while not (srx_host(id).receive(msg)) do
                begin
                    if( srx_host(id).idle ) then
                        activate srx_host(id);
                    passivate;
                end;

                dma_transfer(msg.bytes);
                activate srx_host(id);

                !remove frames from buffer   ;
                return_code := outofbuf(summaryptr.id);
              end;
            end;
        end
        else
        begin
        ! update buffer for CTRL or ACK frames    ;
                return_code := win.rxtcp_outof(frame);
                temp :- frame;

                if (buf.current=/=NONE) then
                        buf.current :- (buf.current).SUC;

                if (temp=/=NONE) then
                begin
                        buf_space := buf_space + temp.bytes;
                        temp.OUT;
                        temp :- NONE;
                end;
        end;
    end
    else
    begin
    ! no frame to receive, passivate                          ;
        state := FREE;
        activate srx_ip(id);
        passivate;
    end--of--if;

    frame :- None;

    ! simulate processing time with hold          ;
    hold(interrupt_time);
    end--of--main;
END++of++RXTCP;
```

```
!                  SANTA CLARA UNIVERSITY

                   DEVELOPED FOR NASA/AMES

                   NCC2-554

                   PERFORMANCE ANALYSIS OF LAN      ;

! IP.SIM - used as include file in level_2.SIM     ;

!---------------------------------------------------------------;
! RX_IP - IP Receiver                                           ;
!                                                               ;
! STATES: FREE, BUSY                                            ;
! ACTIONS: Receives a frame pointer (REF) from its station's    ;
!          rx_csmacd layer, calls the tcp receive function      ;
!          RX_RECEIVE to put the frame into rx_tcp's buffer.    ;
!---------------------------------------------------------------;
ip CLASS rx_ip;
begin
    passivate;                       ! initial startup wait state      ;
    while TRUE do
    begin
        while frame==NONE do         ! if no frame then do nothing     ;
        begin
            state := FREE;
            activate srx_csmacd(id); ! give dependent layer a          ;
                                     ! chance to react to state change ;
            passivate;
        end;
        if frame=/=NONE then          ! frame received from csmacd      ;
        begin
            state := BUSY;
            return_code := FAILED;   ! initialize value of return_code ;
            while (return_code = FAILED) do
            begin                    ! loop until TCP receives the frame ;
                return_code := srx_tcp(id).rx_receive(frame);
                if (return_code = OK) then
                begin
                    activate srx_tcp(id);
                    frame :- NONE;
                end;
                if (return_code = FAILED) then
                begin
                    activate srx_tcp(id);
                end;
            end;
            frame :- NONE;            ! reset the ip receiver parameters ;
            state := FREE;
        end;
        hold(interrupt_time);        ! simulate the processing time of IP;
    end;
end;


!---------------------------------------------------------------;
! TX_IP - IP Transmitter                                        ;
!                                                               ;
! STATES: FREE, SENDING, READY                                  ;
!           tx_tcp layer, calls the csmacd receive function      ;
!           to put the frame into tx_csmacd's buffer.            ;
!----------------------------------------------------------------;
!  CLASS tx_ip;
begin

    procedure reset_tx_ip;
    begin
        frame :- NONE;
        state := FREE;
    end;

    procedure transmit_failed(frameptr);
    REF(frameunit) frameptr;
    begin
        stx_tcp(id).transmit_failed(frameptr);
    end;

begin
    passivate;
    while TRUE do
    begin
        while frame==NONE do          ! state = FREE, no frame to send;
        begin
            passivate;
        end;
                                      ! state = READY, have frame     ;
                                      ! state set in receive procedure;
        !-  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -   ;
        !    Attempt to pass frame to csmacd, if attempt fails then     ;
        !    passivate... will try again when activated by csmacd or    ;
        !    tcp.  Will not succeed until csmacd actually receives the  ;
        !    frame.                                                     ;
        !-  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -   ;
        while (not stx_csmacd(id).receive(frame) ) do
        begin
            passivate;
        end;
        state := SENDING;
        if stx_csmacd(id).IDLE then
        begin
            activate stx_csmacd(id);
        end;
        reset_tx_ip;                      ! frame passed to csmacd - reset;
        activate stx_tcp(id);             ! Give TCP a chance to react to ;
                                          ! IP state change.              ;
        hold(interrupt_time);             ! simulate processing time      ;
    end--while--true--do;
end;
end++TX_IP;
```

```
!               SANTA CLARA UNIVERSITY

                DEVELOPED FOR NASA/AMES

                NCC2-554

                PERFORMANCE ANALYSIS OF LAN      ;
!**********************************************************;
!;
!;
!       process class tx_csmacd;
!;
!       substructure of: class protocol;
!;
!       calls:  ( state routines );
!               csmacd_txs_acquire_channel      ;
!               csmacd_txs_attempt_tx    ;
!               csmacd_txs_idle ;
!               csmacd_txs_wait_for_re_tx       ;
!               csmacd_txs_disabled      ;
!;
!       returns: none ;
!;
!       globals used: ;
!               loops on state and change_status.;
!               this indicates if a change of state has;
!               occurred.;
!
(       actions: mimicks the transmitter of the csmacd ;
!;
!**********************************************************;
        ! declare entities for the csmacd layer;

        csmacd class tx_csmacd;
        begin
                ! attributes;
                integer n;                  ! num of transmission tries;
                long real       tx_delay;       !transmission delay;
                long real       total_delay;    ! total delay;


                ! state routines;
                ! determine if a transition will occur;
                ! return a boolean true if transition occurs;

! files that contain the state subroutines and specific procedures;
%INCLUDE csmacd_txs.sim
%INCLUDE csmacd_tx.sim


                ! main body;
                begin

                        ! initialize attributes;
                        n := 0;

                        ! loop forever;
                        while true do
                        begin
```

```
if (( not change_state ) and
        ( not buffer_interrupt )) then
begin
        passivate;
        change_state := true;
end
else
begin

! reset the transition;
change_state := false;

! identify the state;
if ( state = DISABLED ) then
begin
change_state :=
        csmacd_txs_disabled     ;
end

else if (state = WAIT_FOR_RE_TX ) then
begin
        change_state :=
        csmacd_txs_wait_for_re_tx;

        ! have we waited max tries;
        if ( not change_state ) then
        begin
            ! reset the variables;
            reset_tx_csmacd;

            ! wake up ip;
            activate stx_ip(id);
        end;
end

else if (state = FREE ) then
begin
        change_state :=
        csmacd_txs_idle ;
end

else if (state = ATTEMPT_TX ) then
begin
        change_state :=
        csmacd_txs_attempt_tx    ;
end

else if (state = ACQUIRE_CHANNEL) then
begin
        ! get the frame to ;
        ! send from the buffer;
        frame :-
                get_frame_from_buffer;

        ! if we got the frame ;
        ! send it on ;
        if(frame =/= none ) then
        begin
        change_state :=
                csmacd_txs_acquire_channel(frame);

        ! wake up ip after transmission;
```

```
                               if ( change_state ) then
                               begin
                                       ! calculate total delay;
                                       total_delay :=
                                               prop_delay + tx_delay;

                                   ! activate the csmacd_rx;
                                   activate srx_csmacd(frame.dest_addr);

                               end;
                               end;

                               ! reset the flag;
                               reset_tx_csmacd;

                               ! wake up ip layer;
                               activate stx_ip(id);

                        end

                        else
                        begin
                        outtext(" ERROR IN IDENTIFYING C_TX_STATE");
                        outint(state,5);
                        outimage;
                        end;
                        end;

                end;
        end;
end;
```

```
!                 SANTA CLARA UNIVERSITY

                  DEVELOPED FOR NASA/AMES

                  NCC2-554

                  PERFORMANCE ANALYSIS OF LAN      ;
!*******************************************************;
!;
!;
!                 procedure csmacd_txs_disabled;
!;
!        called by: process class tx_csmacd;
!;
!        calls: csmacd_tx_end_jam;
!;
!        returns: true ; !        transition always occurs;
!;
!        globals used: none ;
!;
!        actions: holds for a jam period;
!                        then calls end jam for a change of state;
!;
!*******************************************************;
boolean procedure csmacd_txs_disabled;
begin
        boolean return_code;               ! true if transition occurs;
        long real jam_time;                        ! amount of time for jam;

        return_code := true;

        ! assign delay time and hold for it;
        jam_time := 2 * tau;
        hold ( jam_time);

        ! call end jam for transition state;
        csmacd_tx_end_jam;

        ! always return true ,since there's always a transition;
        csmacd_txs_disabled := return_code;

end;

!*******************************************************;
!;
!;
!        procedure csmacd_txs_wait_for_re_tx;
!;
!        called by: process class tx_csmacd;
!;
!        calls: csmacd_tx_end_of_delay;
!;
!        returns: true if n <= 16, false otherwise;
!
!        globals used: checks the value of n declared in tx_csmacd ;
!;
!        actions: if n <= 16, calls csmacd_tx_end_of_delay;
!;
!*******************************************************;
boolean procedure csmacd_txs_wait_for_re_tx;
```

```
        boolean return_code;
        integer max_tries;                     ! max number of attempted tx;

        max_tries := 16;


        ! call the transition if n <= max_tries;
        if ( n <= max_tries ) then
        begin
                return_code := csmacd_tx_end_of_delay;
        end
        else
        begin
                ! return frame pointer to ip;
                frame :- get_frame_from_buffer;
                stx_ip(id).transmit_failed(frame);

                return_code := false;
        end;

        csmacd_txs_wait_for_re_tx := return_code;

end;

!************************************************************;
!;
!;
!       procedure csmacd_txs_idle;
!-
!       called by: process class tx_csmacd;
!;
!       calls: csmacd_tx_attempt_send;
!;
!       returns: true if transition occurs, false otherwise;
!;
!       globals used: ;
!               buf.state;
!;
!       actions: represents the idle state;
!;
!************************************************************;
boolean procedure csmacd_txs_idle;
begin
        boolean return_code;

        ! initialize return code;
        return_code := false;

        ! is there a possible transition;
        ! if the buffer is full_tx;
        ! attempt to send;

        ! is there something in the buffer to send;
        if ( buf.state = FULL) then
        begin
                return_code := csmacd_tx_attempt_send;
        end;

        csmacd_txs_idle := return_code;

end;
```

```
!***********************************************************************;
!;
!;
        procedure csmacd_txs_attempt_tx;
!;
!       called by: process class tx_csmacd;
!;
!       calls: csmacd_tx_end_of_collision_window;
!               csmacd_tx_collision;
!;
!       returns: true if transition occurs, false otherwise;
!;
!       globals used: tau ;
!;
!       actions: represents attempting to tramsmit state;
!               checks the channel variable to see if collision;
!               has occurred between attempt_to_send and now;
!               if not, holds for two tau total before checking;
!               to see if collision has occurred. This is the ;
!               collision window time;
!;
!***********************************************************************;
boolean procedure csmacd_txs_attempt_tx;
begin
        boolean return_code;

        return_code := true;

        ! can we proceed;
        ! hold before checking state variable;
        hold(tau);

        if ( schannel_csmacd.csmacd_channel_state  = 0 ) then
        begin
                ! increment the access variable;
                schannel_csmacd.csmacd_channel_state :=
                        schannel_csmacd.csmacd_channel_state + 1;

                ! hold for the collision window;
                hold(tau);

                ! did we collide with another carrier;
                if(schannel_csmacd.csmacd_channel_state  > 1 ) then
                begin
                        csmacd_tx_collision;
                end
                else
                begin
                        csmacd_tx_end_of_collision_window;
                end;
        end
        else
        begin
                ! increment the access variable;
                schannel_csmacd.csmacd_channel_state :=
                        schannel_csmacd.csmacd_channel_state + 1;

                hold(tau);

                csmacd_tx_collision;
        end;
        csmacd_txs_attempt_tx := return_code;

end;

!************************************************************;
!;
!;
!         procedure csmacd_txs_acquire_channel;
!;
!        called by: process class tx_csmacd;
!;
!        calls: csmacd_tx_end_of_tx_message;
!;
!        returns: true if transition occurs, false otherwise;
!;
!        globals used: ;
!                tau;
!;
!        actions: represents the tramsmit state;
!                calculates the tx_delay and the transmit_time;
!;
!************************************************************;
boolean procedure csmacd_txs_acquire_channel(sending_frame);
ref( frameunit) sending_frame;  ! frame to be sent;
begin
        long real transmit_time;
        long real transmit_rate;
        boolean return_code;

        return_code := false;

        ! calculate transmit time;
        tx_delay := sending_frame.bytes * 8 / data_rate;

        transmit_time :=tx_delay - 2 * tau;
        if ( transmit_time < 0 ) then
                transmit_time := 0.0;
        hold ( transmit_time);

        ! is this the end of the transmission message;
        return_code := csmacd_tx_end_of_tx_message(sending_frame);

        csmacd_txs_acquire_channel := return_code;

end;
```

```
!                   SANTA CLARA UNIVERSITY

                    DEVELOPED FOR NASA/AMES

                    NCC2-554

                    PERFORMANCE ANALYSIS OF LAN      ;
!*********************************************************;
!;
!;
!                   procedure csmacd_tx_end_jam;
!;
!       called by: csmacd_txs_disabled ;
!;
!       calls: none;
!;
!       returns: none ;
!;
!       globals used: ;
!               state := WAIT_FOR_RE_TX;
!               schannel_csmacd.csmacd_channel_state:= ;
!                       schannel_csmacd.csmacd_channel_state- 1;
!;
!       actions: ends the jam;
!               transitions the transmitter to wait for re-tx.;
!               changes channel state to free by decrementing states;
!               checks channel_queue with last transmitter in jam;
!
!  *******************************************************;
procedure csmacd_tx_end_jam;
begin
        ! from disabled state;
        ! owns the transition from disabled to wait;
        ! for re-transmission;
        ! (jam over );

        ! reset state variables;
        ! transmitter waits for re_tx;

        state := WAIT_FOR_RE_TX;

        ! decrement the channel;
        schannel_csmacd.csmacd_channel_state:=
                        schannel_csmacd.csmacd_channel_state - 1;

        ! was this the last transmitter in the jam? ;
        if ( schannel_csmacd.csmacd_channel_state = 0 ) then
        begin
                ! check to wake up channel queue;
                csmacd_tx_check_channel_queue;

        end;
end;

!*********************************************************;
!;
!;
!                   procedure csmacd_tx_end_of_delay;
!;
```

```
        called by: csmacd_txs_wait_for_re_tx ;



!;
!       calls: none;
!
        returns: ;
!               true if number of tries is less than sixteen;
!               false if max number of tries attempted;
!;
!       globals used: ;
!               n       number of transmission attempts;
!;
!       actions: transitions to the idle state;
!;
!***********************************************************;
boolean procedure csmacd_tx_end_of_delay;
begin
        boolean return_code;
        long real wait_time;
        integer wait_max;


        ! initialize variables;
        return_code := true;
        n := n + 1;


        ! get a wait max value based on n;
        if ( n > 10 ) then
        begin
                wait_max := 1023;
        end
        else
        begin
                wait_max := ( 2 ** n - 1) - 1;
        end;

        ! get the wait time;
        wait_time := randint (0,wait_max,u );

        hold( wait_time * tau );

        ! reset state variable;
        state := FREE;


        ! check to see if we're over 16 attempts;
        if( n > 16 ) then
        begin
                return_code := false;
        end;

        csmacd_tx_end_of_delay := return_code;

 end;

!***********************************************************;
 !;
 !;
 !                  procedure csmacd_tx_attempt_send;
 !;
 !      called by: csmacd_txs_idle ;
!        calls: none;
!;
!        returns: ;
                always true;
!;
!        globals used: ;
!               state := ATTEMPT_TX;
!               schannel_csmacd.csmacd_channel_state:= ;
!                       schannel_csmacd.csmacd_channel_state + 1;
!               channel_queue;
!;
!        actions: try to gain control of the channel;
!               transitions the transmitter from idle to attempt_tx.;
!               if not possible,goes into channel queue;
!;
!***********************************************************;
boolean procedure csmacd_tx_attempt_send;
begin
        boolean return_code;

        ! from idle state;
        ! owns the transition from idle to ;
        ! attemptimg to send;

        ! initialize variables;
        return_code := true;

        ! is the channel free;
        ! reset state variables;

        ! can we get the channel;
        if ( schannel_csmacd.csmacd_channel_state = FREE ) then
        begin
                ! we can get the channel;
                state := ATTEMPT_TX;

        end
        else
        begin
                ! wait for the channel;
                ! no change of state,passive in queue;
                wait(schannel_csmacd.channel_queue);
        end;


        ! assign return value;
        csmacd_tx_attempt_send := return_code;
end;

!***********************************************************;
!;
!;
!               procedure csmacd_tx_end_of_collision_window;

!        called by: csmacd_txs_attempt_tx ;
!;
!        calls: none;
!;
!        returns: none ;
!;
```

```
!                   state := ACQUIRE_CHANNEL;
!;
        actions: ends the collision window;
                 transitions the transmitter to end of collision window;
!                owns the transition from attempting to ;
!                transmit to acquiring channel;
!;
!*********************************************************;
procedure csmacd_tx_end_of_collision_window;
begin
        ! from attemptimg to transmit state;

        ! reset state variables;
        state := ACQUIRE_CHANNEL;

end;



!*********************************************************;
!;
!;
!                   procedure csmacd_tx_collision;
!;
!       called by: csmacd_txs_attempt_tx ;
!;
!       calls: none;
!;
        returns: none ;
!;
!       globals used: ;
!                   state := DISABLED;
!;
!       actions: ends the jam;
!                   owns the transition from attempting to ;
!                   transmit to global collision;
!                   transitions the transmitter to disabled.;
!;
!*********************************************************;
procedure csmacd_tx_collision;
begin
        ! from attemptimg to transmit state;

        ! reset state variables;
        state := DISABLED;

end;

!*********************************************************;
!;
!;
!                   procedure csmacd_tx_end_of_Tx_frame;

        called by: csmacd_txs_acquire_channel ;
!;
!       calls: none;
!;
!       returns: true if frame received,false otherwise ;
!;
!       globals used: ;
!                           schannel_csmacd.csmacd_channel_state - 1;
!;
!         actions: ends of transmission;
                   calls the rx_csmacd receive routine ;
!                  decrements the channel state variable;
!                  checks the channel queue for entities waiting ;
!                          for the queue;
!;
!**********************************************************;
boolean procedure csmacd_tx_end_of_tx_message(sending_frame);
ref( frameunit) sending_frame;
begin
        boolean return_code;
        return_code := true;

        ! from transmitting  state;
        ! owns the transition from transmitting to ;
        ! idle;

        ! change the receiver message pointer;
        return_code := srx_csmacd(sending_frame.dest_addr).receive(frame) ;

        ! decrement the channel variables;
        schannel_csmacd.csmacd_channel_state:=
                schannel_csmacd.csmacd_channel_state - 1;

        ! do we have anybody in the channel queue;
        csmacd_tx_check_channel_queue;
(
        ! set the return code;
        csmacd_tx_end_of_tx_message := return_code;
end;

!**********************************************************;
!;
!;
!                 procedure csmacd_tx_check_channel_queue;
!;
!         called by: csmacd_tx_end_of_tx_message ;
!;
!         calls: none;
!;
!         returns: none ;
!;
!         globals used: ;
!                 schannel_csmacd.channel_queue ;
!;
!         actions: checks the channel queue to see if anybody ;
!                                 waiting;
!                 takes any transmitters waiting out of the    ;
!                 queue and activates them;
!;
!;
!**********************************************************;
p  ocedure csmacd_tx_check_channel_queue;
begin
        ref( tx_csmacd ) next_transmitter;

        ! wake up anybody  in line for channel;
        if( not (schannel_csmacd.channel_queue.empty )) then
        begin
```

```
                begin
                        next_transmitter :- schannel_csmacd.channel_queue.first;
                        next_transmitter.out;
                        activate next_transmitter;
                end;
        end;
end;


!***********************************************************;
!;
!;
!               procedure reset_tx_csmacd;
!;
!       called by: tx_csmacd ;
!;
!       calls: reset;
!               clear_buffer;
!;
!       returns: none;
!;
!       globals used: ;
!               n := 0;
!               tx_delay := 0;
!               total_delay := 0;
!               change_state:= false;
!               clears the buffer;
!;
!       actions: ;
!               resets the csmacd_tx variables;
!;
!***********************************************************;
procedure reset_tx_csmacd;
begin
        ! reset entity variables;
        reset;

        ! reset the tx_csmacd variables;
        n := 0;
        tx_delay := 0;
        total_delay := 0;
        change_state := false;

        ! clear the buffer;
        clear_buffer;

end;

procedure dma_transfer(no_bytes);
!-------------------------------------------------------------;
! procedure dma_transfer                                      ;
!                                                             ;
! This procedure executes a hold to simulate a dma transfer.;
! Tt used the number of bytes passed to it to determine the  ;
! actual lenght of the hold.                                  ;
!                                                             ;
! Globals: dma_xfer_rate                                      ;
!-------------------------------------------------------------;
integer no_bytes;
begin
    hold(no_bytes * 8 /dma_xfer_rate);
```

```
!***********************************************************;
!
!
!        boolean procedure receive;
!;
!    called by: csmacd_txs_idle,csmacd_rxs_idle ;
!;
!    calls: none;
!;
!    returns: ;
!        true if frame entered into buffer;
!        false otherwise;
!;
!    globals used: ;
!        buf - current buffer;
!        buffer_interrupt - set when frame enters buffer;
!;
!    actions: ;
!        takes a frame and puts it into the buffer;
!        sets the buffer state to full;
!        sets buffer_interrupt to true;
!        holds for dma_xfer time;
!;
!***********************************************************;
boolean procedure receive(in_frame);
ref(frameunit) in_frame;
begin
    boolean return_code;
    return_code := false;

    ! if the buffer isn't full,put the frame in;
    if ( not ( buf.state = FULL )) then
    begin
        ! put the frame in the buffer ;
        in_frame.into(buf);

        ! set the buffer state;
        buf.state := FULL;

        ! set the buffer interrupt;
        buffer_interrupt := true;

        ! hold for transfer time;
        dma_transfer( in_frame.bytes );

        return_code := true;
    end;
    receive := return_code;
end;
```

```
!                   SANTA CLARA UNIVERSITY

                    DEVELOPED FOR NASA/AMES

                    NCC2-554

                    PERFORMANCE ANALYSIS OF LAN      ;
!************************************************************;
!;
!;
!       process class rx_csmacd;
!;
!       substructure of: class protocol;
!;
!       calls: ;
!               csmacd_rxs_idle ;
!               csmacd_rxs_receiving    ;
!;
!       returns: none ;
!;
!       globals used: ;
!               loops on the state and change status boolean
!;
!       actions: mimicks the receiver of the csmacd ;
!               activates rx_ip when appropriate;
!;
!************************************************************;
(       csmacd class rx_csmacd;
        begin

                ! attributes;
                long real       rx_delay;  ! in usecs;


                ! state routines;
                ! determine if a transition will occur;
                ! return a boolean true if transition occurs;

! files that contain the state subroutines and specific procedures;
%INCLUDE csmacd_rxs.sim
%INCLUDE csmacd_rx.sim


                ! main body;
                begin
                        rx_delay := 10.0;


                        ! initialize attributes;

                        ! loop forever;
                        while true do
                        begin

                                ! do we check for a transition ;
                                if (( not change_state ) and
                                        ( not buffer_interrupt )) then
                                begin
                                        passivate;
                                        change_state := true;
                                        else
                                        begin

                                        ! reset the transition;
                                        change_state := false;

                                        ! identify the state;
                                        if (state = FREE ) then
                                        begin
                                        change_state :=
                                                  csmacd_rxs_idle ;
                                        end

                                        else if (state = RECEIVING ) then
                                        begin
                                        change_state :=
                                                  csmacd_rxs_receiving     ;

                                                  ! is the buffer full receiving;
                                                  if ( change_state ) then
                                                  begin
                                                            ! wake up the ip layer;
                                                            activate srx_ip(id);

                                                            ! reset the variables;
                                                            reset_rx_csmacd;
                                                  end;
                                        end

                                        else
                                        begin
                                        outtext(" ERROR IN IDENTIFYING C_RX_STATE");
                                        outint(state,5);
                                        outimage;
                                        end;
                                        end;

                              end;
                    end;
          end;
```

```
!               SANTA CLARA UNIVERSITY

                DEVELOPED FOR NASA/AMES

                NCC2-554

                PERFORMANCE ANALYSIS OF LAN     ;
!*****************************************************************;
!;
!;
!       procedure csmacd_rxs_idle;
!;
!       called by: process class rx_csmacd;
!;
!       calls: csmacd_rx_frame;
!;
!       returns: true if transition occurs, false otherwise;
!;
!       globals used: buf.state ;
!;
!       actions: represents the idle state;
!;
!*****************************************************************;
boolean procedure csmacd_rxs_idle;
begin
        boolean return_code;

(       ! initialize return code;
        return_code := false;

        ! is there a possible transition;
        ! attempt to send;

        if (  buf.state = FULL ) then
        begin
                return_code := csmacd_rx_frame;
        end;

        csmacd_rxs_idle := return_code;
end;


!*****************************************************************;
!;
!;
!       procedure csmacd_rxs_receiving;
!;
!       called by: process class rx_csmacd;
!;
!       calls: csmacd_rx_buffer_full;
!;
!       returns: true if transition occurs, false otherwise;
,
.       globals used: none ;
!;
!       actions: represents receiving state;
!               checks to see if the buffer's full;
!;
!*****************************************************************;
boolean procedure csmacd_rxs_receiving;
```

```
        boolean return_code;
        return_code := false;

        ! check to see if we have a transition;
        return_code := csmacd_rx_buffer_full;

        ! hold for a receiving time;
        hold(rx_delay);

        csmacd_rxs_receiving := return_code;

end;
```

```
!                   SANTA CLARA UNIVERSITY

                    DEVELOPED FOR NASA/AMES

                    NCC2-554

                    PERFORMANCE ANALYSIS OF LAN        ;
!***********************************************************;
!;
!;
!                   procedure csmacd_rx_frame;
!;
!       called by: csmacd_rxs_idle ;
!;
!       calls: none;
!;
!       returns:  always true ;
!;
!       globals used: none ;
!;
!       actions: changes the state from idle to receiving ;
!               owns the transition for recieving message;
!;
!***********************************************************;
boolean procedure csmacd_rx_frame;
begin
        boolean return_code;
        return_code := true;

        ! reset state variables;
        state := RECEIVING;

        ! set the return value;
        csmacd_rx_frame := return_code;
end;

!***********************************************************;
!;
!;
!                   procedure csmacd_rx_buffer_full;
!;
!       called by: csmacd_rxs_receiving ;
!;
!       calls: none;
!;
!       returns: none ;
!               true if frame was sent to rx_ip;
!               false otherwise;
!;
!       globals used: frame     ( frame pointer ) ;
!;
!       actions: imitates the receiving state ;
!               owns the transition for recieving buffer full;
!***********************************************************;
boolean procedure csmacd_rx_buffer_full;
begin
        boolean return_code;

        ! initialize variables;
        ! get the next frame to send;
        ! or loop on the last one;
        if ( frame == none ) then
        begin
                frame :- get_frame_from_buffer;
        end;

        ! if we got the sending frame, pass it up;
        if ( frame =/= none ) then
        begin
                if( srx_ip(id).state = FREE ) then
                begin
                        ! set the frame pointer for the ip layer;
                        return_code := srx_ip(id).receive(frame) ;
                end
                else
                begin
                        return_code := false;
                end;
        end
        else
        begin
                return_code := false;
        end;

        ! set the return value;
        csmacd_rx_buffer_full := return_code;
[illegible]d;

!*********************************************************;
!;
!;
!               procedure reset_rx_csmacd;
!;
!       called by: rx_csmacd ;
!;
!       calls: reset;
!               clear_buffer;
!;
!       returns: none;
!;
!       globals used: ;
!               change_state:= false;
!               clears the buffer;
!;
!       actions: ;
!               resets the rx_csmacd variables;
!;
!*********************************************************;
procedure reset_rx_csmacd;
begin

        ! reset entity variables;
        reset;

        ! reset the rx_csmacd variables;
        change_state := false;

        ! clear the buffer;
        clear_buffer;
```

```
end;

  ɔcedure dma_transfer(no_bytes);
!---------------------------------------------------------------------;
! procedure dma_transfer                                              ;
!                                                                     ;
! This procedure executes a hold to simulate a dma transfer.;
! It used the number of bytes passed to it to determine the ;
! actual lenght of the hold.                                          ;
!                                                                     ;
! Globals: dma_xfer_rate                                              ;
!---------------------------------------------------------------------;
integer no_bytes;
begin
    hold(no_bytes * 8 /dma_xfer_rate);
end;



!***********************************************************;
!;
!;
!       boolean procedure receive;
!;
!   called by: csmacd_txs_idle,csmacd_rxs_idle ;
!;
!_  calls: none;
(
:   returns: ;
!       true if frame entered into buffer;
!       false otherwise;
!;
!   globals used: ;
!       buf - current buffer;
!       buffer_interrupt - set when frame enters buffer;
!;
!   actions: ;
!       takes a frame and puts it into the buffer;
!       sets the buffer state to full;
!       sets buffer_interrupt to true;
!       holds for dma_xfer time;
!;
!***********************************************************;
boolean procedure receive(in_frame);
ref(frameunit) in_frame;
begin
    boolean return_code;
    return_code := false;

    ! if the buffer isn't full,put the frame in;
    if ( not ( buf.state = FULL )) then
    begin
        ! put the frame in the buffer ;
        in_frame.into(buf);

        ! set the buffer state;
        buf.state := FULL;

        ! set the buffer interrupt;
        buffer_interrupt := true;
        ! hold for transfer time;
        dma_transfer( in_frame.bytes );

        return_code := true;
    end;
    receive := return_code;
end;
```

```
!                 SANTA CLARA UNIVERSITY

                  DEVELOPED FOR NASA/AMES

                  NCC2-554

                  PERFORMANCE ANALYSIS OF LAN     ;
!**********************************************************;
!;
!;
!       process class channel_csmacd;
!;
!       substructure of: class protocol;
!;
!       calls: none;
!;
!       returns: none ;
!;
!       globals used: initializes channel variables ;
!       csmacd_channel_state:= 0 ( FREE );
!       channel_queue :- new head;
!;
!       actions: mimicks the channel ;
!;
!**********************************************************;
class channel_csmacd;

  gin

        ! attributes;
        integer csmacd_channel_state;

        ref (head) channel_queue;


        ! main body;
        ! initialize attributes;
        begin
                csmacd_channel_state:= 0;

                channel_queue :- new head;
        end;

end;
```

Type [illegible]

```
10.0      ! data_rate;
1.0       ! interrupt_time;
40.0      ! dma_xfer_rate;
  )000040        ! aver_arrival_time;
5.0       ! prop_delay;
1500      ! max_frame_size;
5000      ! aver_msg_size;
```

type b

```
10.0     ! data_rate;
0.5      ! interrupt_time;
80.0     ! dma_xfer_rate;
  )000040         ! aver_arrival_time;
5.0      ! prop_delay;
1500     ! max_frame_size;
5000     ! aver_msg_size;
```

Output. 1D- a

```
station #   throughput                  aver delay / frame  success  reject
    1       6.716818915&-001            1.861000000&+003      14       0
    2       5.179557857&-001            2.413333405&+003      14       0
    3       7.370051365&-001            1.696053308&+003      10       0
    4       7.934756301&-001            1.575347689&+003      14       0
    5       6.274293987&-001            1.992256025&+003      12       0
    6       7.085492589&-001            1.764168100&+003      10       0
    7       1.832410092&-001            6.821617089&+003      12       0
    8       5.897607711&-001            2.119503469&+003      16       0
    9       7.829448668&-001            1.596536427&+003      16       0
   10       1.042457252&-001            1.199089937&+004      14       0
  network_throughput =      1.948454521&-001
  aver_delay_per_frame =      3.383071488&+003
  simulation time      3.387300000&+006
```

output. 10.0

```
station #   throughput              aver delay / frame success reject
    1       5.537486600&-001        2.257341806&+003     20      0
    2       7.157506226&-001        1.746418320&+003     28      0
    3       8.632726815&-001        1.447978173&+003     16      0
    4       2.485822573&-001        5.028516571&+003     20      0
    5       7.053833527&-001        1.772086051&+003     10      0
    6       6.988605855&-001        1.788625694&+003     15      0
    7       7.574594085&-001        1.650253447&+003     26      0
    8       7.150107464&-001        1.748225473&+003     19      0
    9       8.180771125&-001        1.527973318&+003     17      0
   10       7.873636565&-001        1.587576452&+003     16      0
  network_throughput =      1.860952051&-001
  aver_delay_per_frame =      2.055499530&+003
  simulation time      5.024310000&+006
```

out put. mix

```
station #   throughput                aver delay / frame  success  reject
    1       6.9214948958-001          1.8059682478+003       19       0
    2       1.4205486428-001          8.7994170908+003       10       0
    3       1.2776807428-001          9.7833516508+003       14       0
    4       1.8811536238-001          6.6448586918+003       18       0
    5       6.9047403348-001          1.8103504838+003       18       0
    6       7.7613669738-001          1.6105410358+003       19       0
    7       8.1485481538-001          1.5340156028+003       16       0
    8       8.0594557568-001          1.5509732148+003       14       0
    9       3.5157386798-001          3.5554405898+003       21       0
   10       8.7234549068-001          1.4329185098+003       12       0
  network_throughput =      1.8642285808-001
  aver_delay_per_frame =      3.8527835118+003
  simulation time      4.3181400008+006
```